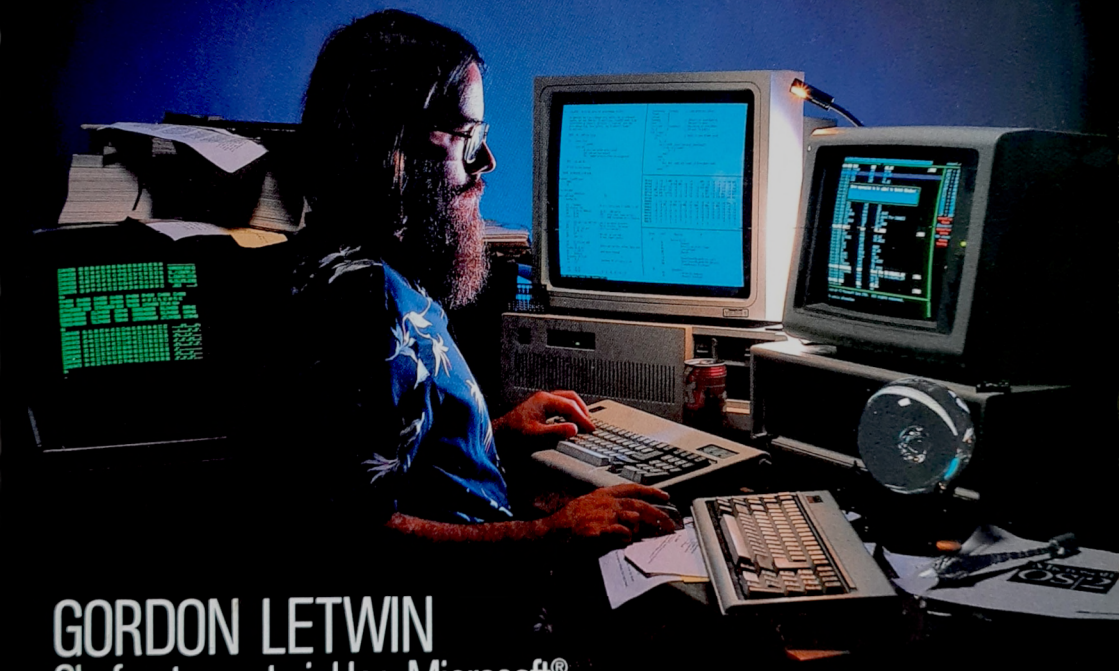
INSIDE OS/2

BESCHREIBUNG · EINSATZ · TECHNISCHE DETAILS



GORDON LETWIN
Chefsystementwickler, Microsoft®

Microsoft PRESS®
VIEWEG

Geleitwort von Bill Gates

INSIDE OS/2

BESCHREIBUNG · EINSATZ · TECHNISCHE DETAILS

GORDON LETWIN

INSIDE OS/2

BESCHREIBUNG · EINSATZ · TECHNISCHE DETAILS

GORDON LETWIN
Chefsystementwickler, Microsoft®

Übersetzt von Andreas Dripke und Angelika Schätzel

Microsoft PRESS®
VIEWEG

Geleitwort von Bill Gates

CIP-Titelaufnahme der Deutschen Bibliothek

Letwin, Gordon:
Inside OS 2: Beschreibung, Einsatz, techn. Details/Gordon Letwin. Übers. von Andreas Dripke u. Angelika Schätzel. Geleitw. von Bill Gates. – Redmond, Washington: Microsoft Press; Braunschweig: Vieweg, 1988
Einheitssacht.: Inside OS 2 (dt.)
ISBN 3-528-04628-7

Das Buch ist die deutsche Übersetzung von
Gordon Letwin, Inside OS/2
Microsoft Press, Redmond, Washington 98073
© 1988

Übersetzung aus dem Amerikanischen von
Andreas Dripke und Angelika Schätzel, Wiesbaden

Das in diesem Buch enthaltene Programm-Material ist mit keiner Verpflichtung oder Garantie irgendeiner Art verbunden. Der Autor, die Übersetzer und der Verlag übernehmen infolgedessen keine Verantwortung und werden keine daraus folgende oder sonstige Haftung übernehmen, die auf irgendeine Art aus der Benutzung dieses Programm-Materials oder Teilen davon entsteht.

Der Verlag Vieweg ist ein Unternehmen der Verlagsgruppe Bertelsmann.

Alle Rechte vorbehalten
© Friedr. Vieweg & Sohn Verlagsgesellschaft mbH, Braunschweig 1988

Das Werk einschließlich aller seiner Teile ist urheberrechtlich geschützt. Jede Verwertung außerhalb der engen Grenzen des Urheberrechtsgesetzes ist ohne Zustimmung des Verlags unzulässig und strafbar. Das gilt insbesondere für Vervielfältigungen, Übersetzungen, Mikroverfilmungen und die Einspeicherung und Verarbeitung in elektronischen Systemen.

Druck und buchbinderische Verarbeitung: Lengericher Handelsdruckerei, Lengerich
Printed in Germany

ISBN 3-528-04628-7

Inhaltsverzeichnis

Vorwort XI
Einführung XV

Teil 1 Die Aufgabe 1

1 Projektgeschichte 3
1.1 MS-DOS Version 1.0 5
1.2 MS-DOS Version 2.0 6
1.3 MS-DOS Version 3.0 8
1.4 MS-DOS Version 4.0 8

2 Ziele und Kompatibilitätsüberlegungen 11
2.1 Ziele 11
2.1.1 Graphische Benutzerschnittstelle 12
2.1.2 Multitasking 16
2.1.3 Speicherverwaltung 17
2.1.4 Schutz 19
2.1.5 Isolierung (Encapsulation) 20
2.1.6 Kommunikation zwischen Prozessen – interprocess communication (IPC) 21
2.1.7 Direkter Zugriff auf Einheiten 23
2.2 Kompatibilitätserwägungen 24
2.2.1 Real-Modus gegen Protect-Modus 24
2.2.2 Ablauf von Anwendungen im Real-Modus (Kompatibilitätsmodus) 25
2.2.2.1 Speicherausnutzung 27
2.2.2.2 Dateisperre 28
2.2.2.3 Netzwerk-Überlegungen 29
2.2.2.4 Kompatibilität häufig benutzter Funktionen 29
2.2.3 Abwärtskompatibilität 30
2.2.3.1 Family API 31
2.2.3.2 Netzwerkkompatibilität: Server-Benutzer 31

3 Strukturprinzipien für OS/2 33
3.1 Maximale Flexibilität 34
3.2 Eine stabile Umgebung 38

3.2.1 Speicherschutz ... 38
3.2.2 Schutz vor Nebeneffekten ... 42
3.3 Fehlersuche ... 45
3.4 Software-Tools ... 48

Teil 2 Die Architektur ... 51

4 Multitasking ... 53
4.1 Das Modell der Subtasks ... 57
4.1.1 Standarddateinummern ... 58
4.1.2 Anonyme Pipes ... 64
4.1.3 Weitere Details ... 68
4.2 PIDs und Befehlsunterbäume ... 74
4.3 DosExecPgm ... 77
4.4 DosCWait ... 83
4.5 Steuerung von Kind-Tasks und Befehlsunterbäumen ... 87
4.5.1 DosKillProcess ... 88
4.5.2 DosSetPrty ... 88

5 Threads und Scheduler-Prioritäten ... 89
5.1 Threads ... 89
5.1.1 Thread-Stapel ... 90
5.1.2 Verwendung von Threads ... 91
5.1.2.1 Vorder- und Hintergrundarbeit ... 91
5.1.2.2 Asynchrone Verarbeitung ... 92
5.1.2.3 Beschleunigte Ausführung ... 93
5.1.2.4 Programmstruktur ... 93
5.1.3 Zu berücksichtigende Grenzen ... 94
5.1.3.1 Lokale Variablen ... 95
5.1.3.2 RAM-Semaphoren ... 96
5.1.3.3 DosSuspendThread ... 96
5.1.3.4 DosEnterCritSec/DosExitCritSec ... 97
5.1.4 Thread 1 ... 97
5.1.5 Beenden von Threads ... 98
5.1.6 Durchsatz ... 99
5.2 Scheduler/Priorität ... 100
5.2.1 Allgemeine Prioritätskategorie ... 101
5.2.1.1 Unterkategorie Hintergrund ... 102
5.2.1.2 Unterkategorien Vordergrund und interaktiv ... 103
5.2.1.3 CPU- oder Ein-/Ausgabebindung ... 104

5.2.2 Kategorie Zeitkritische Priorität 106
5.2.3 Kategorie Niedrige Priorität 107
5.2.4 Setzen von Prozeß-/Thread-Prioritäten 108

6 Die Benutzerschnittstelle 109
6.1 VIO-Benutzerschnittstelle 110
6.2 Benutzerschnittstelle des Presentation Manager 110
6.3 Presentation Manager und VIO-Kompatibilität 112

7 Dynamische Verknüpfung 113
7.1 Statische Verknüpfung 113
7.2 Dynamische Verknüpfung beim Laden 115
7.3 Dynamische Verknüpfung zur Laufzeit 119
7.4 Dynamische Verknüpfung, Prozesse und Threads 121
7.5 Daten 122
7.5.1 Instanzdaten 126
7.5.2 Globale Daten 127
7.6 Dynlink-Pakete als Unterroutinen 128
7.7 Unterroutinen 129
7.7.1 Unterstützung für Untersysteme 130
7.8 Dynlinks als Prozeßschnittstellen 133
7.9 Dynlinks als Schnittstellen zum Kernel 136
7.10 Dynlinks als Teil der Betriebssystemarchitektur 138
7.11 Details der Implementierung 140
7.11.1 Dynlink-Datensicherheit 140
7.11.2 Dynlink: Nutzung gemeinsamer Ressourcen 141
7.11.3 Nebeneffekte von Dynlink-Modulen 146
7.12 Dynlink-Namen 148

8 Dateiverwaltung 149
8.1 Dateinamen 149
8.2 Netzwerkzugriff 152
8.3 Namensgenerierung und Kompatibilität 152
8.4 Berechitungen 154
8.5 Andere Objekte im Dateinamensbereich 154

9 Speicherverwaltung 157
9.1 Schutzmodell 157
9.2 Schnittstelle (API) zur Speicherverwaltung 161
9.2.1 Gemeinsam benutzter Speicher 161
9.2.2 Große Speichersegmente (Huge Memory) 164
9.2.3 Ausführung aus Datensegmenten 168
9.2.4 Speicherunterbelegung 169

9.3 Speicherauslagerung ... 175
9.3.1 Hinweise zur Auslagerung ... 180
9.4 Status und Information ... 184

10 Umgebungsstrings ... 187

11 Kommunikation zwischen Prozessen ... 193
11.1 Gemeinsam benutzter Speicher ... 193
11.2 Semaphoren ... 194
11.2.1 Wiedergewinnung von Semaphoren ... 199
11.2.2 Zuordnung der Semaphoren ... 202
11.3 Benannte Pipes ... 202
11.4 Warteschlangen ... 211
11.5 Dynamischer Datenaustausch DDE ... 214
11.6 Signale ... 215
11.7 Verknüpfung von IPC-Formen ... 216

12 Signale ... 217

13 Presentation Manager und VIO ... 223
13.1 PM oder VIO? ... 225
13.2 Hintergrundein-/-ausgabe ... 228
13.3 Graphik unter VIO ... 228

14 Interaktive Programme ... 237
14.1 Ein-/Ausgabearchitektur ... 238
14.2 Ctrl-C und Ctrl-Break ... 245

15 Das Dateiverwaltungssystem ... 251
15.1 Das OS/2-Dateiverwaltungssystem ... 253
15.2 Verwaltung der Datenträgernamen ... 255
15.3 Effizienz bei der Ein- und Ausgabe ... 261

16 Überwachung, Datenintegrität, Zeitgeberroutinen ... 263
16.1 Kontrolle der Geräteeinheiten ... 264
16.2 Datenintegrität ... 269
16.2.1 Semaphoren ... 272
16.2.2 DosBufReset ... 276
16.2.3 Durchschreiben ... 277
16.3 Zeitgeberroutinen ... 279

17 Einheitentreiber und Hardwarefehler ... 283
17.1 Einheitentreiber ... 283
17.1.1 Einheitentreiber und OS/2-Kommunikation ... 285
17.1.2 Modell zur Programmierung eines Einheitentreibers ... 287

17.1.3 Einheitenverwaltung 297
17.1.4 Dual-Modus 297
17.2 Hardwarefehler 299
17.2.1 Der Hardwarefehler-Daemon 301
17.2.2 Hardwarefehlerbehandlung durch Anwendungen . 303

18 Ein-/Ausgabeprivilegmechanismus und Debugging/Ptrace 305
18.1 Ein-/Ausgabeprivilegmechanismus 305
18.2 Debugging/Ptrace 308

19 Privilegierte Funktionen und die 3X-Box 313

20 Family API 319

Teil 3 Die Zukunft 327

21 Die Zukunft 329
21.1 Dateisystem 330
21.2 Der 80386 336
21.2.1 Large-Segmente 336
21.2.2 Mehrere Real-Modus-Boxen 337
21.2.3 Vollständiger Schutz 337
21.2.4 Andere Leistungsmerkmale 338
21.3 Die nächsten zehn Jahre 339

Glossar 341
Sachwortverzeichnis 359

Vorwort

Das Betriebssystem OS/2 wurde mit der Intention entwickelt, ein Stück Softwaregeschichte zu schreiben. In den kommenden zehn Jahren werden Millionen von Programmierern und Anwendern mit dem System arbeiten. Dabei werden unweigerlich von Zeit zu Zeit Fragen auftreten, warum das eine Leistungsmerkmal fehlt oder warum ein anderes vorhanden ist. Die Antworten auf diese Fragen lassen sich am besten verstehen, wenn man die OS/2 zugrundeliegenden Ideen und Konzepte - die Produktphilosophie - kennt. Das vorliegende Buch legt diese Philosophie offen. Der Autor Gordon Letwin ist der Chefentwickler bei Microsoft, der die Architektur von OS/2 geschaffen hat. Letwin legt auf den folgenden Seiten ausführlich dar, warum er sich entschieden hat, die einen Leistungsmerkmale in das Betriebssystem aufzunehmen, während andere weggelassen wurden.

Die erste Mikrocomputergeneration umfaßte 8-bit-Systeme wie den Commodore Pet, den TRS-80 oder den Apple II sowie die Geräte auf der Basis von CPM/80. In praktisch alle Systeme war der BASIC-Interpreter von Microsoft eingebaut. Ich traf Gordon Letwin zum erstenmal, als ich die PC-Abteilung von Heath (heute zu Zenith gehörend) besuchte. Letwin hatte sowohl ein eigenes BASIC als auch ein Betriebssystem für den Heath-Rechner geschrieben und war alles andere als glücklich, daß das Management von Heath erwog, ein externes BASIC zu lizensieren. In einer Gruppe von etwa 15 Personen legte Letwin die Nachteile meiner BASIC-Version im Vergleich zu seiner dar. Nachdem Heath mein BASIC genommen hatte, konnte ich Letwin überzeugen, daß Microsoft das richtige Unternehmen ist, großartiger Software zum Marktdurchbruch zu verhelfen. Letwin wurde einer der ersten zehn Programmierer bei Microsoft. Sein erstes Projekt war es, alleine einen Compiler für Microsoft-BASIC zu schreiben. Er versah seine Bürotür mit einem Schild, auf dem stand

Das Tier bitte nicht stören, füttern, anfassen ...,

und nach fünf Monaten hatte Letwin einen exzellenten Compiler entwickelt, der noch heute die Grundlage für alle unsere BASIC-Compiler darstellt. Im Gegensatz zu einer Vielzahl von Starprogrammierern schreibt Letwin Quellcode, der ein herausragendes Beispiel für Lesbarkeit darstellt und präzise Erklärungen aller Algorithmen sowie Begründungen für die Wahl der Algorithmen umfaßt.

Als Intel den Mikroprozessor 80286 auf den Markt brachte, wußten wir nicht, wie es möglich sein sollte, den Protect-Modus der CPU, der neben dem kompatiblen Modus zur Verfügung stand, zu nutzen. Wir hatten bereits aufgegeben, als Gordon Letwin die im vorliegenden Buch beschriebene Idee kam, das zu tun, was er als "Abstellen des Autos bei 100 km pro Stunde" bezeichnet. Als wir diese Idee Intel und deren Kunden präsentierten, glaubte keiner, daß die Methode funktionieren würde. Selbst Letwin war nicht sicher, bis er mit einigen Testprogrammen feststellen konnte, daß seine Idee tatsächlich richtig war.

Die Aufgabe von Gordon Letwin als Betriebssystemarchitekt besteht darin, unsere Konzepte zu überblicken und dafür zu sorgen, daß sie so einfach und so elegant wie möglich sind. Dazu gehört auch die Kontrolle über den von anderen Programmierern verfaßten Code. Die meisten Entwickler mögen es, wenn Letwin ihren Code inspiziert und Verbesserungsvorschläge unterbreitet. Oft genug ist das Resultat ein um die Hälfte gekürztes Programm. In mindestens einem Fall allerdings brach der Programmierer in Tränen aus, nachdem er die Anmerkungen von Letwin gesehen hatte. Letwin kontrolliert nämlich nicht nur den Code der Entwickler, sondern greift handfest ein, wo es ihm nötig erscheint. Kürzlich hat er sich entschlossen, den größten Teil unseres neuen Dateiverwaltungssystems selbst zu schreiben. Das Ergebnis: Das neue System arbeitet drastisch schneller als das alte. In einem "Urlaub" vor einiger Zeit schrieb Letwin mehr als 50 Seiten Quellcode.

Bei dem vorliegenden Buch handelt es sich um das Erstlingswerk von Gordon Letwin. Wie jeder gute Designer hat er sich bereits ausgemalt, was Kritiker an Negativem über das Buch sagen werden. Nach meiner Überzeugung bietet das Werk Lesefreude. Noch wichtiger aber: Es ist ein bedeutsames Buch. Ich hoffe, Sie werden es genauso genießen wie ich.

Bill Gates

Einführung

Ein technologischer Durchbruch unterscheidet sich signifikant von stufenweisen technischen Verbesserungen. Beim stufenweisen Fortschreiten - der Verbesserung einer bestehenden Sache - nimmt die Entwicklung einen geradlinigen Verlauf ohne Überraschungen. Die Verbesserungen werden vorgenommen, ihr Nutzen wird erkannt und umgesetzt.

Ein wesentlicher Fortschritt, der weitgehend für sich alleine steht - also ein technologischer Durchbruch - spielt sich anders ab. Die Kommunikation stellt hierfür ein gutes Beispiel dar. Zu Anfang dieses Jahrhunderts existierte eine weitumspannende Infrastruktur für die Kommunikation zwischen den Menschen. Die Post wurde noch zweimal am Tag zugestellt, zudem gab es die Möglichkeit der Nachrichtenübermittlung durch Kurierdienste.

Der Geschäftsmann hat die Nachricht seiner Sekretärin diktiert, diese hat sie an das nächstgelegente Kurierbüro weitergereicht, von dem sie zum Empfänger gebracht wurde. Die Erfindung des Telefons stellte einen technologischen Durchbruch dar, nicht nur eine technische Verbesserung, da sie die Art und Weise änderte, in der Menschen miteinander kommunizieren. Bezeichnend für diesen Durchbruch, wie für andere auch, war die Tatsache, daß er zunächst auf Unverständnis stieß. Viele Geschäftsleute schätzten den praktischen Nutzen des Telefons als gering ein. "Wo liegt der Vorteil? Ich muß die Nachricht diktieren, meine Sekretärin muß sie niederschreiben und in den Telefonraum weiterreichen, von dort ist der Empfänger anzurufen, die Nachricht muß übermittelt werden - möglicherweise sogar fehlerhaft - sie ist an die Sekretärin des Empfängers weiterzureichen - in dieser Zeit hat meine Nachricht den Empfänger auch per Kurierdienst erreicht. Dies gilt umso mehr, als die meisten meiner Geschäftspartner räumlich nahe tätig sind. Zudem spare ich bei der Kurierdienstlösung die Kosten für eine Person, die den ganzen Tag am Telefon sitzt und auf ankommende Nachrichten wartet."

Es handelt sich um das klassische Beispiel eines technologischen Durchbruchs, der von den Menschen zunächst in seiner Bedeutung überhaupt nicht erkannt wurde. Die potentiellen Nutzer versuchten, die neue Technologie in ihre vorhandenen Organisationsschemata zu pressen. Der Geschäftsmann erkannte nicht, daß es mit der Erfindung des Telefons völlig überflüssig geworden war, die Nachrichten einer Sekretärin zu diktieren und daß sie auch beim Empfänger nicht mehr zu Papier gebracht werden mußten. Er erkannte auch nicht, daß die Tatsache, daß seine Geschäftspartner räumlich nah gelegen sind, lediglich eine Folge der Nachrichtenübermittlung per Kurierdienst war. Durch das Telefon wurden diese Notwendigkeiten auf einmal überflüssig. Man konnte die Büroräume nach anderen Gesichtspunkten, etwa der Nähe zu Produktionsanlagen oder der Höhe des Mietspiegels, auswählen. Die immensen Vorteile des Telefons konnte allerdings erst genutzt werden, nachdem sich ein allgemeiner Konsens über den Einsatz der neuen Technologie gebildet hatte. Das Schlüsselwort heißt *Akzeptanz*.

Ein wesentliches Hindernis bei der Akzeptanz neuer Technologien stellt die fehlende neue Infrastruktur dar. Der Nutzen eines Telefons ist gleich null, solange der Adressat einer Nachricht nicht ebenfalls über eines verfügt. Es liegt in der Natur des Telefonidee, daß es einen allgemein akzeptierten Standard geben muß. Es nützt nichts, wenn der Adressat einer Nachricht zwar ein Telefon besitzt, wenn dieses nicht an das gleiche Leitungsnetz angeschlossen ist wie das des Senders. Hinzu kam damals, daß die Technologie des Telefons noch ganz am Anfang stand und die Geräte somit umständlich zu handhaben waren.

Dennoch konnten alle diese Umstände den Siegeszug des Telefons nicht aufhalten. Die Kommunikationsbedürfnisse einiger weniger Menschen waren derart stark, daß sie Möglichkeiten fanden, diese Nachteile auszugleichen oder zu umgehen. Andere setzten die neue Technologie aus Kühnheit oder Ehrgeiz ein, um ihr bestehendes System zu vergrößern. Wieder andere nutzten schließlich die enorme Leistungsfähigkeit der neuen Technologie zur Verbesserung der bestehenden Technologien. So mag manches Kurierserviceunternehmen geografisch verteilte Büros

eingerichtet und diese per Telefon miteinander verbunden haben. Die Geschwindigkeit bei der Übermittlung kurzer Nachrichten ließ sich auf diese Weise beträchtlich reduzieren, indem die Meldungen fernmündlich durchgesagt und niedergeschrieben wurden. Der Einsatz des Telefons in dieser Weise ist natürlich absurd, aber solange die Nachfrage nach herkömmlicher Kurierübermittlung kräftig genug war, mußte jede Verbesserung willkommen sein, auch wenn sie "absurd" war.

Hat eine neue Technologie erst einmal den Fuß in der Tür, läuft sie unaufhaltsam weiter. Nach einiger Zeit entstehen Standards, die Fehler werden bereinigt und - am wichtigsten - die neuen Möglichkeiten verändern das Verhalten des Anwender. Nachdem das Telefon verfügbar war, änderte sich das Geschäftsleben und später auch das Privatleben der Menschen in einer Weise, die zuvor nicht denkbar gewesen wäre. Natürlich waren die Kurierdienste zu ihrer Zeit schnell genug, aber eben nur, solange das Telefon noch nicht existierte. Sie waren nämlich nur deshalb schnell genug, weil es einfach keine schnellere Möglichkeit der Nachrichtenübermittlung gab.

Veränderungen dieser Art können bewirken, daß ein intelligenter Mensch erklärt "das Telefon ist ein albernes Spielzeug" und einige Jahre später sagt "das Telefon ist unverzichtbar". Diese Veränderungen des Technikanwenders - durch die Technik selbst ausgelöst - machen es extrem schwierig, die Auswirkungen einer neuen Technologie vorauszusagen. Die Extrapolation gegenwärtiger Trends führt in die Irre, weil die neue Technologie viele Grundannahmen der Extrapolation falsch werden läßt, während die in der Zukunft gültigen Annahmen noch nicht abzusehen sind. Dies erklärt im übrigen, warum uns die meisten Zukunftsvisionen aus früheren Zeiten heute lächerlich und absurd erscheinen, obgleich sie durchweg von intelligenten und gebildeten Menschen stammen. Ihr einziger, aber gravierender Fehler bestand darin, daß sie einen technologischen Durchbruch irrtümlich als eine Verbesserung bestehender Technologien ansahen. Sie konnten somit voraussagen, welchen Einfluß die neue Technologie auf das bestehende geschäftliche und private Verhalten der Menschen haben würde, aber sie konnten nicht die

künftigen, gänzlich veränderten Verhaltensweisen der Menschen erfassen.

Digitalrechner stellen ein typisches Beispiel für einen technologischen Durchbruch dar. Sie haben die klassischen drei Schritte bei der Einführung einer neuen Technologie überwunden: "exotische Spielzeuge", "begrenzter Nutzen" und schließlich "unentbehrlich". Großrechner sind diese Straße vom Anfang bis zum Ende gegangen, sowohl im kommerziellen als auch im technisch/wissenschaftlichen Umfeld. IBM hatte den Markt für Computer zunächst auf einige Dutzend Systeme veranschlagt. Mit der Technologie und der unterstützenden Infrastruktur sowie der Anpassung der Menschen an die Computer stieg jedoch auch der Nutzen und damit das Einsatzfeld der Rechner - vom staatlichen Volkszählungsbüro über Versicherungsgesellschaften und die Lohnabrechnung bis hin zu völlig neuen Einsatzgebieten wie Managementinformations- (MIS) oder Flugbuchungssystemen.

Mikrocomputer durchlaufen einen ähnlichen Entwicklungsprozeß. Das Stadium "exotisches Spielzeug" haben sie bereits hinter sich gelassen und sind nun beim "begrenzten Nutzen" angelangt. Sie werden in wenigen Jahren das dritte Stadium - die Unentbehrlichkeit - erreichen. Für dieses dritte Stadium hat Microsoft OS/2 konzipiert und geschaffen.

Obgleich die Analyse der Entwicklung im Großrechnerbereich eine gute Grundlage für die Beurteilung des Mikrocomputerszenarios ist, kann man nicht davon ausgehen, daß Mikrocomputer einfach "preiswerte Großrechner" seien. Die Mikrocomputerrevolution wird den bei einem technologischen Durchbruch üblichen Weg gehen, der sich nicht unmittelbar aus der historischen Entwicklung bei Großcomputern ableiten läßt.

Das vorliegende Werk wurde geschrieben, um den Tatsachen gerecht zu werden, daß die Mikrocomputerentwicklung einen technologischen Durchbruch darstellt, und daß uns die zweite industrielle Revolution bevorsteht. Die Entwickler von OS/2 haben versucht, das System so zu entwickeln, daß es den Anforderungen in höchstem Maße gerecht wird, die mit dem Stadium

der Unentbehrlichkeit von Mikrocomputern aufkommen. Um dies zu verstehen, genügt es nicht, Kenntnisse über MS-DOS zu haben oder die Handbücher zu OS/2 genau zu studieren. Dies hängt nicht nur mit der Komplexität des Systems zusammen, sondern insbesondere auch damit, daß in OS/2 viele Konzepte realisiert sind, die aus dem Blickwinkel vor dem technologischen Durchbruch als unnötig erscheinen. OS/2 bietet teilweise Lösungen für Probleme, die noch gar nicht aufgekommen sind, sondern erst mit der breiten Akzeptanz von Mikrocomputern Bedeutung gewinnen.

Es gibt andere Bücher, die konkrete Informationen und Anleitungen zur Programmierung unter OS/2 geben. Im vorliegenden Werk werden die grundlegenden architektonischen Modelle von OS/2 erklärt. Zudem wird dargelegt, wie diese Architekturkonzepte in eine Büroumgebung der Zukunft mit veränderten Anforderungen passen werden. Der Schwerpunkt liegt dabei auf den grundlegenden Aspekten, Problemen und Lösungen, die unabhängig davon wichtig sind, in welcher Programmiersprache und in welcher Schnittstellenumgebung ein Programmierer seine Software entwickelt.

Wie bei vielen technischen Objekten hängen die einzelnen Elemente von OS/2 durchweg durch die eine oder andere Beziehung zusammen. Bei der Gliederung des Buches wurde eine Einteilung nach logischen Gesichtspunkten soweit als möglich vorgenommen. Dabei ließen sich jedoch im Sinne der Klarheit gewisse Redundanzen nicht vermeiden. Manchmal ist es einfach erforderlich, einen Sachverhalt an mehreren Stellen unter unterschiedlichen Blickwinkeln darzulegen. Im Sachwortverzeichnis sind alle Referenzstellen eines Schlagwortes ausgeführt, wichtige und weniger wichtige. Bei der Suche nach Informationen zu einem bestimmten Thema empfiehlt es sich, zunächst im Inhaltsverzeichnis die entsprechenden Kapitel auszuwählen und in einem zweiten Schritt aus dem Sachwortverzeichnis Detailinformationen zusammenzusuchen, die sich möglicherweise in anderen Kapiteln befinden.

Teil 1
Die Aufgabe

Kapitel 1

Projektgeschichte

Die Firma Microsoft wurde in der Erwartung gegründet, irgendwann werde es auf jedem Schreibtisch einen Mikrocomputer geben. Diese Vorstellung schien einer zweiten industriellen Revolution gleichzukommen, nachdem die erste industrielle Revolution die körperliche Arbeit mechanisiert hatte (bis dahin hatte der Mensch sich allein auf seine Muskelkraft verlassen müssen, abgesehen von einfachen Hilfsmitteln wie zum Beispiel dem Ackergaul, der Wagen und Pflug hinter sich herzog). Die zweite industrielle Revolution nun sollte Abläufe geistiger Arbeit mechanisieren. Und obwohl wir heute nahe daran sind, diese Revolution realisiert zu haben, so ist es doch immer noch der Mensch, der geduldig "Kopfarbeit" zu leisten hat.

Einige Aufgaben - die sehr arbeitsaufwendig sind und einem starren Muster gehorchen (wie beispielsweise Lohnabrechnungen) - wurden automatisiert. Der Großteil der Denkarbeit wird aber immer noch von Menschen und nicht von Computern erledigt (wir haben sozusagen nur das Äquivalent für den Ackergaul, nicht aber für die Schlagbohr- oder Waschmaschine).

Natürlich können Computer nicht die Kreativität (zumindest in naher Zukunft) des Menschen ersetzen, wie Maschinen auch nicht das Design oder die Kreativität im körperlichen Bereich ersetzt haben. Der größte Teil der Arbeit in Büros besteht allerdings aus routinemäßiger Manipulation von Informationen. Die zweite industrielle Revolution wird uns von dieser "Knochenarbeit" befreien (routinemäßige Datenmanipulation, Analyse und Hilfestellungen zur Entscheidungsfindung), damit der Mensch sich auf die wirklich schwierigen Aufgaben konzentrieren kann.

Viele Menschen sehen die Unausweichlichkeit der zweiten industriellen Revolution nicht. Sie erkennen nicht, wie ein Computer 75 % ihrer Arbeit erledigen kann, weil diese ohne Vorhandensein des Computers strukturiert wurde. Aber, wie immer bei technologischen Durchbrüchen, so wird auch dieses Mal die neue Technik die Menschen und ihre Art, mit ihr umzugehen, verändern.

Beispielsweise wird ein Großteil der Arbeit nur schwerlich auf den Computer übertragbar sein, da Daten in Form von Ausdrucken erhalten werden und die Eingabe zu aufwendig wäre. Vor zehn Jahren glaubte man noch, daß dereinst ein papierloses Büro die Lösung des Problems darstellen sollte: Alle Daten und Materialien sollten vom Computer erzeugt und ausschließlich auf elektronischem Wege übertragen werden. Büros werden sicherlich in Zukunft mit weniger Papier auskommen und die Verfügbarkeit von leistungsfähigen Netzwerken wird die Entwicklung beschleunigen. Trotzdem bleibt das Papier ein wertvolles Hilfsmittel zur Datenübertragung. So hat die Entwicklung eine neue Richtung gezeigt: Informationen auf Papier werden inzwischen vom Computer nicht nur ausgegeben, sondern auch angenommen. Leistungsfähige Laserdrucker, Desktop-Publishing-Systeme und Scanner sowie die optische Zeichenerkennung werden in Zukunft die Ein-/Ausgabe in Verbindung mit Papier immer weiter perfektionieren.

Obwohl die Gründer von Microsoft die Auswirkungen der zweiten industriellen Revolution durchdacht und verstanden haben, kann niemand mit Sicherheit voraussagen, wie die Zukunft aussehen wird. Aus diesem Grunde basieren die täglich von Microsoft getroffenen Entscheidungen auf zwei Arten von Zielen: *kurzfristigen Zielen*, die bekannt sind, und *langfristigen Zielen*, die der Vorstellung eines vollautomatisierten Büros treu bleiben. Die Entscheidungen richten sich an den kurzfristigen Zielen aus, müssen aber auch mit den langfristigen Zukunftsvorstellungen konform gehen, deren genauere Realisierung sich erst im Laufe der Revolution herauskristallisieren kann.

Bei der Ankündigung der 16-bit-Mikroprozessoren wußte Microsoft, daß die Zeit reif genug war, um der Verwirklichung ihrer Zukunftsvision näherzukommen. Eine leistungsfähige Computerumgebung basiert auf einem ausgewogenen Gerät und einem hochentwickelten Betriebssystem. Die Hardware war nun verfügbar und das Standardbetriebssystem für die 8-bit-Mikroprozessoren war nicht mehr adäquat. An diesem Punkt begann Microsoft mit der Entwicklung von Betriebssystemen: Wir wußten, daß ein leistungsfähiges Betriebssystem benötigt wurde, und es gab nur eine Möglichkeit, dies zu garantieren: Eigenentwicklung.

1.1 MS-DOS Version 1.0

Die Geschichte von MS-DOS begann damit, daß Microsoft von IBM mit der Entwicklung eines Diskettenbetriebssystems beauftragt wurde, welches für die neue Generation der IBM-Computer verwendet werden sollte, dem *IBM Personal Computer* (PC). Microsofts einziges Betriebssystem war zu diesem Zeitpunkt XENIX, eine lizensierte Version des AT&T-Betriebssystems UNIX. UNIX/XENIX läuft nur auf einem Prozessor mit Speicherverwaltung und Schutzleistungsmerkmalen. Da der 8086-/8088 über keines dieser Leistungsmerkmale verfügte und die Speichererfordernisse für XENIX/UNIX für damalige Standards extrem hoch waren, mußte ein anderes Betriebssystem entwickelt werden.

CP/M-80 von Digital Research Incorporated (DRI) war der Standard für ein 8-bit-Betriebssystem, und der Großteil der existierenden Mikrocomputersoftware wurde unter CP/M-80 entwickelt. Aus diesem Grunde entschloß sich Microsoft, die MS-DOS-Version 1.0 mit CP/M-80 so kompatibel wie möglich zu halten. Der 8088-Prozessor konnte existierende CP/M-80-Programme, die für den 8080 entwickelt waren, nicht ausführen, aber die althergebrachten Programme konnten einfach und halbautomatisch für die Verwendung auf dem 8088 konvertiert werden. Microsoft war sich bewußt, daß eine Minimierung der

Anpassungsprobleme zur schnelleren Akzeptanz ihres neuen Betriebssystems MS-DOS für den IBM PC führen würde.

Ein so grundlegendes neues Software-Produkt erfordert viel Entwicklungszeit, und IBM wollte seinen PC schnell auf den Markt bringen. Aus diesem Grund suchte Microsoft auswärts nach Software, auf der das neue Betriebssystem MS-DOS aufgebaut werden konnte. Ein solches Produkt fand sich bei der Seattle Computer Products. Tim Paterson, ein Softwareentwickler, hatte einen *CP/M-80-Clone* erzeugt, den er SCP-DOS nannte und der auf dem 8088 ablauffähig war. Microsoft kaufte alle Rechte an diesem Produkt und dessen Quellcode und verwendete es als Ausgangsbasis für die Entwicklung von MS-DOS Version 1.0.

MS-DOS Version 1.0 wurde im August 1981 freigegeben. Nur für den IBM PC verfügbar, bestand es aus 4000 Zeilen Assemblercode und benötigte 8 Kbytes Speicherkapazität. MS-DOS Version 1.1 wurde 1982 freigegeben und besaß als grundlegendes neues Leistungsmerkmal die Unterstützung doppelseitiger 320-Kbyte-Disketten.

Microsofts Ziel bestand darin, MS-DOS Version 1.0 zu CP/M in hohem Maße kompatibel zu halten. Ironischerweise war es zu dem alten Betriebssystem kompatibler als das DRI-eigene Produkt CP/M-86. Wie wir noch sehen werden, führte diese CP/M-Kompatibilität für viele Microsoft-Softwareentwickler zu großen Problemen.

1.2 MS-DOS Version 2.0

Anfang 1982 wurde Microsoft von IBM informiert, daß die Entwicklung eines Personal Computers in Auftrag war, der mit Festplatten arbeitet: der IBM XT. Microsoft begann die Arbeit an einer MS-DOS Version 2.0, die die neue Hardware unterstützen sollte. MS-DOS war für eine Diskettenumgebung entwickelt worden, und aufgrund der Kompatibilität zu CP/M-80 waren einige grundlegende Änderungen notwendig. Eine Diskette konnte dazumal nur ein Verzeichnis enthalten, das maxi-

mal 64 Dateien enthalten konnte. Dies war ein vernünftiges Verhältnis, wenn man berücksichtigt, daß eine Diskette über nur 180 Kbytes Speicherkapazität verfügte.

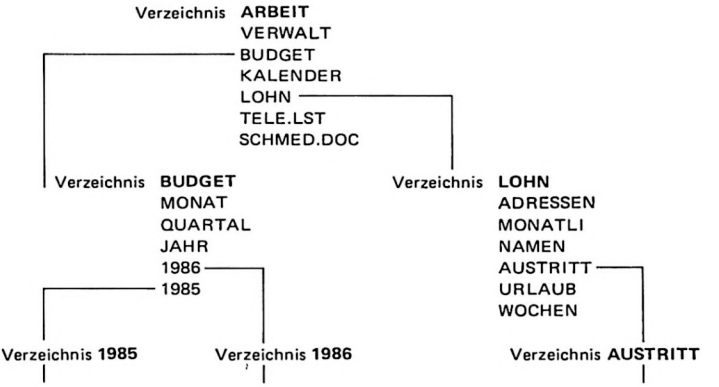


Bild 1.1 Eine baumartige Verzeichnishierarchie. Im Verzeichnis ARBEIT findet man die Dateien VERWALT.TXT, KALENDER, SCHED.DOC und TELE.LST sowie die zwei Unterverzeichnisse (BUDGET, LOHN). Jedes Unterverzeichnis besitzt wiederum Unterverzeichnisse.

Für eine Festplatte war die Begrenzung von 64 Dateien allerdings wesentlich zu niedrig und die Verwendung eines einzigen Verzeichnisses zur Verwaltung von Hunderten von Dateien unzumutbar. Aus diesen Gründen fügten die Entwickler der zweiten Version von MS-DOS (Mark Zbikowski, Aaron Reynolds, Chris Peters und Nancy Panners) ein hierarchisches Dateiverwaltungssystem ein. In einem solchen Dateiverwaltungssystem kann ein Verzeichnis weitere Verzeichnisse (Unterverzeichnisse) und Dateien enthalten. Die Unterverzeichnisse können wiederum Dateien und Verzeichnisse enthalten usw. Ein hierarchisches System beginnt mit dem *Stammverzeichnis* (*Hauptverzeichnis*),

das eine baumähnliche Ansammlung von Dateien und Verzeichnissen (Bild 1.1) enthalten kann.

1.3 MS-DOS Version 3.0

MS-DOS 3.0 wurde im August 1984 eingeführt, zu dem Zeitpunkt, an dem IBM den IBM AT ankündigte. Der AT enthält einen anderen Prozessor, den 80286, und verwendet einen Emulationsmodus für den 8086, wenn er unter DOS betrieben wird. In diesem Modus ist der 80286 nichts anderes als ein schneller 8086. Der erweiterte Adressierungsbereich des Bausteins und der *Protect-Modus* werden bei der DOS-Emulation nicht verwendet.[1]

MS-DOS Version 3.1 wurde im November 1984 freigegeben und bot die Möglichkeit der Unterstützung von Netzwerken. Im Januar 1986 war MS-DOS Version 3.2 (eine verbesserte Version 3.1) verfügbar. Diese Version unterstützte erstmals 3,5-Zoll-Disketten und enthielt die Formatierfunktion für Geräteeinheiten im Einheitentreiber. Im Jahr 1987 folgte MS-DOS Version 3.3. Der wesentliche Vorteil dieser Version bestand in der Unterstützung des IBM PS/2 und kompatibler Hardware.

1.4 MS-DOS Version 4.0

Im Januar 1983 begann Microsoft die Arbeit an einer Multitasking-Version des MS-DOS. Zu diesem Zeitpunkt wurde diese Version noch MS-DOS Version 3.0 genannt. Durch die Verfügbarkeit einer weiteren MS-DOS Version (die nicht multitaskingfähig war) unter dem Namen *MS-DOS Version 3.0* wurde die Multitasking-Version intern zu MS-DOS Version 4.0. Eine Version des Produkts (multitaskingfähig, aber im *Real-Modus* ablaufend) wurde als MS-DOS Version 4.0 vermarktet. Da die

---

1. Produkte wie Microsoft XENIX/UNIX sind unter Verwendung des *Protect*-Modus auf dem PC/AT und Kompatiblen ablauffähig (da diese und ähnliche Betriebssysteme keine Vorgängerversionen im *Real*-Modus besitzen).

Version 4.0 von MS-DOS nur im *Real-Mode* läuft, kann sie mit den Prozessoren 8088, 8086 und 80286 betrieben werden. Die Beschränkungen, die der *Real-Modus* mit sich bringt, machen MS-DOS Version 4.0 aber zu einem sehr spezialisierten Produkt. Obwohl MS-DOS 4.0 ein mit Vorrechten versehenes Multitasking voll unterstützt, ist der Systemspeicher im *Real*-Modus auf 640 Kbyte begrenzt. Eine virtuelle Speichernutzung ist nicht gegeben; das bedeutet, daß alle Prozesse im Speicherbereich von 640 Kbyte ohne Speicherauslagerung[2] ablaufen müssen. Nur eine mit MS-DOS Version 3 kompatible Anwendung kann im *Real*-Modus betrieben werden; die anderen Prozesse müssen speziell auf MS-DOS Version 4.0 abgestimmt sein, ihre Umgebung interpretieren und friedlich mit der MS-DOS Version-3r-Anwendung zusammenarbeiten können.

Wegen dieser Beschränkungen wurde MS-DOS Version 4.0 nicht im üblichen Sinne für den Verkauf freigegeben, sondern war als Basis für bestimmte OEMs zur Unterstützung erweiterter PC-Architekturen gedacht. So konnte beispielsweise eine leistungsfähige Telefonverwaltung in einen PC eingebaut werden. Zur Steuerung der Telefonverbindungen konnte beispielsweise ein Hintergrundprozeß unter MS-DOS Version 4.0 herangezogen und das resultierende System als kompatibler MS-DOS-3-PC mit eingebautem "*Supertelefon*" vermarktet werden.

Trotz der Freigabe von MS-DOS Version 4.0 als spezielles OEM-Produkt wurde das Multitasking-Projekt weitergeführt. Es hieß nun MS-DOS Version 5.0. Das Ziel bestand in der Ausschöpfung der Vorteile des *Protect-Modus* des 80286, um eine allgemein verwendbare Multitasking-Umgebung ohne die Begrenzungen von MS-DOS Version 4.0 zu erstellen. Microsoft und IBM schlossen sich zusammen, um das gesteckte Ziel zu erreichen. Kurz vor der Beendigung des Projektes traf die Marketingabteilung eine Entscheidung: Das neue Produkt sollte

---

2. Es ist übrigens nicht möglich, die Auslagerung von Speicherbereichen ohne Speicherverwaltungshardware (die für den 8086 im Real-Modus nicht erhältlich ist) zu verwirklichen.

OS/2 heißen. Aus diesem Grund wird von nun ab das Design und die Funktion des neuen Betriebssystems *OS/2* beschrieben.

Kapitel 2

Ziele und Kompatibilitätsüberlegungen

OS/2 gleicht den traditionellen Multitasking-Betriebssystemen in mancherlei Weise: Multitasking, Scheduling, Diskettenverwaltung und Speicherverwaltung werden bereitgestellt. Auf der anderen Seite unterscheidet es sich von traditionellen Systemen, da ein Personal Computer sich bereits in seiner Struktur von einem Mehrbenutzer-Minicomputer unterscheidet. Die Designer von OS/2 hatten zweierlei im Blick: direkt anzusteuernde Ziele und Kompatibilitätserwägungen. In diesem Kapitel wird beschrieben, welches diese Ziele und Kompatibilitätserwägungen waren. Auf diese Weise soll der Zusammenhang transparent werden, der für das an späterer Stelle behandelte OS/2-Konzept maßgeblich ist.

2.1 Ziele

Das primäre Ziel von OS/2 ist die Verfügbarkeit eines idealen Büroautomationsbetriebssystems. Die Designer verfolgten dieses Ziel durch die Definition der folgenden, teilweise widersprüchlich scheinenden Aufgabenstellungen:

- Bereitstellung eines geräteunabhängigen Graphiktreibers ohne signifikanten Overhead.
- Direkter Zugriff von Anwendungsprogrammen auf Peripherieeinheiten, wobei die Fähigkeit zur Virtualisierung oder Aufteilung der Verwendung dieser Geräte gegeben sein muß.

- Bereitstellung des Leistungsmerkmals *Multitasking* ohne Minderung der Performance oder der Antwortfähigkeit des Singletasking-Systems.
- Bereitstellung einer angepaßten Umgebung für jedes Programm und dessen aufgerufene Programme; ferner die Bereitstellung einer Standardumgebung, die von anderen Programmen im System nicht verändert werden kann.
- Bereitstellung einer geschützten Umgebung zur Sicherstellung der Systemstabilität, allerdings ohne Beschränkung der Anwendung, wie dies bei nichtgeschützten Systemen der Fall ist.

2.1.1 Graphische Benutzerschnittstelle

Der Mensch erhält am einfachsten und schnellsten Informationen durch das Auge. Die Augen suchen sich die gewünschten Informationen und richten ihre Aufmerksamkeit auf diese Stelle. Ein großer Teil des menschlichen Gehirns dient der Verarbeitung visueller Informationen. Die Aufnahme abstrakter Daten und die Bedeutung visuellen Materials (von Text über Graphiken zu bewegten Bildern) werden hundertmal schneller als alle anderen Arten von Informationen aufgenommen.

Wenn also etwa ein Computersystem große Informationsmengen schnell darstellen können soll, müssen leistungsfähige Graphikmerkmale bereitstehen. In früheren Betriebssystemen für Minicomputer waren diese Leistungsmerkmale sehr beschränkt, da sie viel Speicherkapazität beanspruchen und die Rechenzeit zur Darstellung hochauflösender Graphiken enorm ist. Heute besitzen bereits Mikrocomputer den nötigen Speicher zur Erstellung von Bildschirmdarstellungen; Rechenzeit (CPU-Zeit) ist billig und es gibt keine bessere Verwendung für diese Leistungsmerkmale als die Unterstützung graphischer Anwendungen.

Graphiken können viele Formen annehmen (Bilder, Tabellen, Zeichnungen und Charts); eventuell kann die Farbgebung und Animation hinzugenommen werden. Dies alles sind Verbesserungen in der Textdarstellung. Graphikdarstellungen müssen

nicht notwendigerweise Charts und Bilder enthalten. Das Prinzip des *WYSIWYG* (*What You See Is What You Get*) wird heute immer geläufiger. Durch die Darstellung eines Textes im Graphikmodus können verschiedene Schriftarten und -größen, Proportionalschriften und ähnliches so angezeigt werden, wie sie auch im Ausdruck erscheinen.

Die Bildschirmgraphikkomponente von OS/2 sollte geräteunabhängig sein; das heißt, eine Anwendung sollte in der Lage sein, ein Bild auf einer Peripherieeinheit auszugeben, ohne die Charakteristika der angesprochenen Bildschirmkarte zu kennen. Die verfügbaren Geräte werden jedes Jahr besser und es wäre kurzsichtig, eine Anwendung an einen bestimmten Anzeigeadapter zu binden.

Die Idee besteht in der Verschiebung des einheitenspezifischen Codes in ein Softwarepaket namens *Einheitentreiber*, das mit jeder physikalischen Einheit ausgeliefert wird. Vom Anwendungsprogramm werden allgemeingültige Befehle gegeben, und der Einheitentreiber übersetzt diese in die Charakteristika des angesprochenen Gerätes. Das Resultat dieses Prinzips ist, daß ein Graphikkartenhersteller lediglich einen Einheitentreiber entwickeln und diesen mit seiner Einheit ausliefern muß. Das Anwendungsprogramm muß nicht wissen, um welche Einheit es sich handelt, und der Einheitentreiber muß keine Information bezüglich der ablaufenden Anwendung besitzen. Lediglich die verbindende Schnittstelle muß bekannt sein. Die allgemein gehaltene Schnittstelle beschreibt eine virtuelle Anzeigeeinheit; die allgemeine Technik des Verbergens komplizierter Strukturen hinter einer einfachen, standardisierten Schnittstelle wird *Virtualisierung* genannt.

In Bild 2.1 sehen Sie die traditionelle Struktur eines Einheitentreibers des Betriebssystems. Der Einheitentreiber kann nicht direkt von der Anwendung angesprochen werden, da Einheitentreiber im *Privilege*-Modus des Prozessors ausgeführt werden. Unerwünschte Manipulationen an der korrespondierenden Einheit werden dadurch ausgeschlossen. Das aufrufende Programm muß im Benutzermodus ablaufen. In der Sprache der 80286-

/80386-Familie wird der *Privilege*-Modus auch als *Ring 0* bezeichnet; der normale Modus (Benutzermodus) zum Ablauf von Anwendungsprogrammen ist *Ring 3*. Das Betriebssystem verhält sich wie ein Mittelsmann: Es nimmt eine Anforderung an, prüft diese auf ihre Gültigkeit und verteilt Anforderungen, wenn nur ein Gerät bereitsteht und dieses gleichzeitig von mehreren Anwendungen angesprochen wird. Schließlich wird die Anforderung an den Einheitentreiber übergeben. Die Antwort des Einheitentreibers läuft auf diesem Pfad zurück und erreicht das Anwendungsprogramm über den Umweg des Betriebssystems.

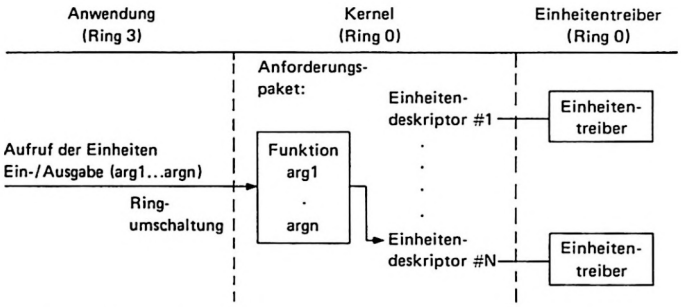


Bild 2.1 Die Architektur traditioneller Einheitentreiber. Zur Ausführung einer Ein-/Ausgabeoperation wird das Betriebssystem aufgerufen, das ein Request-Paket aufbaut und nach der Bestimmung des Zielgerätes dieses übergibt. Die Antwort des Einheitentreibers folgt der Route in umgekehrter Richtung über den Kernel zur Anwendung.

Dieser Ansatz löst das Virtualisierungsproblem leider auf Kosten der Performance. Die Schnittstelle zwischen Anwendung und Einheitentreiber stellt einen Datenübertragungsengpaß dar, so daß die Übergabe eingeschränkt ist. Üblicherweise wird von der Anwendung erwartet, daß von ihr ein *Request*-Block mit al-

len benötigten Daten aufgebaut wird. Der eigentliche Aufruf an das Betriebssystem hat lediglich die Aufgabe, den Request-Block an den Einheitentreiber zu übergeben. Die Initialisierung eines *Request*-Blocks benötigt Zeit, und die Zerlegung im Einheitentreiber ebenfalls. Weitere Zeit wird durch die Antwort benötigt: Der Einheitentreiber baut die Antwort auf, die das Betriebssystem kopiert und die anschließend durch die Anwendung wieder zerlegt wird. Weitere Zeit wird von internen Betriebssystemaufrufen zur Prüfung und zum Kopieren des *Request*-Blockes benötigt (hierunter fallen auch die Festlegung des korrekten Einheitentreibers usw.). Auch das Umschalten zwischen verschiedenen Ringen (*Privilege-* und Benutzermodus) benötigt Zeit; bei jeder Anfrage treten zwei Umschaltungen auf.

Diese Performance-Minderung war bei Systemen ohne Graphik akzeptabel, da die Aktualisierung einer Bildschirmdarstellung üblicherweise 1920 (oder weniger) Bytes benötigte. Heutige Graphikeinheiten benötigen die Bearbeitung von bis zu 256.000 Bytes und mehr pro Aktualisierung des Bildschirms, und zukünftige Einheiten werden noch umfangreichere Datenmengen erfordern. Zudem kann es vorkommen, daß Anwendungen hochauflösende Bildschirmdarstellungen mehrmals in der Sekunde aktualisieren müssen.

Hinweis: Üblicherweise liegt die niedrige Verarbeitungsgeschwindigkeit traditioneller Einheitentreiber nicht in der Menge der übertragenen Daten, sondern in der Anzahl der Anfragen und Antworten begründet. In Verbindung mit Diskette neinheiten ist das althergebrachte Modell vollkommen ausreichend, da hier bei jedem Übertragungsvorgang mit großen Datenmengen (beispielsweise 40.000 Bytes) gearbeitet wird. Anzeigegeräte arbeiten üblicherweise mit kleineren Datenmengen (einem Zeichen, einem Wort oder einer Zeile). In diesem Fall ist die Anzahl der Aufrufe der Faktor, der für die geringe Übertragungsgeschwindigkeit verantwortlich ist.

Für OS/2 brauchte man eine leistungsfähige, geräteunabhängige graphische Anzeigeunterstützung mit einer umfassenden und effizienten Benutzerschnittstelle, die weder Ringumschaltungen noch das Betriebssystem beanspruchen sollte. Wie wir noch sehen werden, entspricht OS/2 diesen Anforderungen durch die Verwendung einer Methode, die *dynamische Verknüpfung* (*dynamic linking*) genannt wird.

2.1.2 Multitasking

Der neue Personal Computer sollte ein Leistungsmerkmal besitzen, das man allgemein *Multitasking* nennt. Menschen führen das Prinzip des Multitasking ständig aus. So könnten sie beispielsweise gleichzeitig an drei Projekten arbeiten, eine Novelle lesen und Spanischunterricht nehmen. Jede dieser Aufgaben erfordert einen Teil ihrer verfügbaren Zeit, wobei jeweils alle Projekte (außer einem) stillgelegt werden. Dies wird *serielles Multitasking* genannt. Menschen können aber auch *paralleles Multitasking* ausführen (zum Beispiel das Fahren eines Autos mit gleichzeitiger Unterhaltung mit dem Mitfahrer).

In einer seriellen Multitasking-Umgebung kann ein Benutzer zwischen verschiedenen Prozessen umschalten. So kann ein Textverarbeitungssystem unterbrochen (*nicht verlassen!*), eine Kalkulationstabelle berechnet und dann zum wartenden Textverarbeitungsprogramm zurückgekehrt werden. Beim Telefonieren kann ein Computerkalender aufgerufen werden, wobei die gerade bearbeitete Kalkulationstabelle in den Hintergrund geschoben wird. Nachdem die Benutzung des Kalenders abgeschlossen ist, wird die Berechnung der Kalkulationstabelle wieder aufgenommen.

Multitasking unter OS/2 ist eine weitere Hauptanforderung: Viele Programme oder Anwendungen laufen zum gleichen Zeitpunkt ab. Das Multitasking dient aber zu mehr als nur dem schnellen Umschalten zwischen Anwendungen: Paralleles Multitasking erlaubt die Abarbeitung eines Prozesses (zum Beispiel den Ausdruck einer umfangreichen Datei oder die Neuberechnung einer Kalkulationstabelle), während der Benutzer mit einer anderen Anwendung arbeitet. Da OS/2 multitaskingfähig ist, können andere Programme zusammen mit der vom Benutzer gerade eingesetzten Anwendung ablaufen; aus diesem Grunde ist OS/2 auch beim Einsatz in Netzwerken (Überwachung der Elektronischen Post im LAN) dienlich.

Hinweis: Die heutigen Geräte enthalten üblicherweise nur eine CPU, so daß immer nur ein einziges Programm aktiv sein kann. Auf einer mikroskopischen Ebene betrachtet, ist OS/2 ein serielles Multitasking-System. Vom Benutzerstandpunkt aus ist das serielle Multitasking allerdings nicht erkennbar, da eine vorausschauende

Umschaltung und Zeitvergabe an die einzelnen Programme realisiert wurde. OS/2 kann die Arbeit der CPU zu jedem Zeitpunkt unterbrechen und die Zentraleinheit einem anderen Programm zuweisen, um dieses weiter zu bearbeiten. Da die Umschaltung mehrmals in der Sekunde vorkommt, ist - zumindest vom Anwenderstandpunkt - das System als ein quasi paralleles Multitasking-System anzusehen.

2.1.3 Speicherverwaltung

Die Multitasking-Fähigkeit ist unter OS/2 nicht allzu schwer zu nutzen. Dies ist bereits durch periodisch auftretende Hardwareinterrupts möglich (zum Beispiel auf der Basis des Systemtakts). Um zu einem leistungsfähigen Programmsystem für Anwendungsprogramme zu gelangen, ist neben der eigentlichen Multitasking-Fähigkeit aber ein effektives Speicherverwaltungssystem für die einzelnen Programme nötig. Wenn beispielsweise zwei Anwendungen auf einem System ohne besondere Speicherorganisation ablaufen, beginnt jede Anwendung in einem niedrigen Speicherbereich mit der Ablage von Daten, um möglichst viel Speicher zur Verfügung zu haben. Durch die Multitasking-Unterstützung - hier ohne besondere Organisationsform - versuchen beide Anwendungen, auf den gleichen Speicherbereich zuzugreifen und zerstören einander.

Ein Speicherverwaltungssystem löst dieses Problem durch den Einsatz einer speziellen Hardware, die in den Prozessoren 80286 und 80386 (zum Beispiel im IBM AT und Kompatiblen sowie in Geräten, die auf dem 80386 basieren) eingebaut ist. Das Speicherverwaltungssystem verwendet die Hardware zur Virtualisierung des Hauptspeichers, so daß jedem Programm scheinbar der gesamte Speicher zur Verfügung steht.

Hinweis: Die Prozessoren 8086 und 8088, die im IBM PC/XT verwendet werden, sind zur Auslagerung von Speicherbereichen nicht fähig, da diese Hardwarevoraussetzung fehlt. Dies ist der Grund, weshalb frühere Versionen von MS-DOS das Multitasking nicht unterstützen und OS/2 auf diesen Maschinen nicht ablauffähig ist.

Für jeden verwendeten Speicherbereich muß bekannt sein, welche Anwendung ihre Daten darin ablegt hat. Gleichermaßen müssen vom Speicherverwaltungssystem nicht mehr benötigte Speicherbereiche (eine Anwendung wurde beispielsweise beendet und hat den ihr zugeordneten Speicherbereich nicht freigegeben) freigegeben werden. Die Annahme, daß alle Anwendun-

gen den von ihnen belegten Speicherbereich nach ihrer Beendigung freigeben, ist unrealistisch. So muß die Speicherorganisation als Bestandteil des Betriebssystems berücksichtigt werden. Da eine Vielzahl von Programmen, die von verschiedenen Herstellern stammen, mit OS/2 ablaufen sollen, wurde eine besondere Art der Speicherverwaltung gewählt.

Ein weiteres Problem muß vom Betriebssystem gelöst werden können: Dem Anwendungsprogramm muß ausreichend Hauptspeicherkapazität zur Verfügung gestellt werden. Obwohl RAM jedes Jahr billiger wird, wachsen die Speicheranforderungen der Anwendungen in überdimensionalem Maße. OS/2 muß demnach eine effektive Methode zur Belegung von mehr Speicher bereitstellen, als physikalisch im System - Intel-Prozessor und Speicherbausteine - vorhanden ist.

OS/2 hat dieses Problem mit Hilfe einer Speicherauslagerungstechnik gelöst. Jedes Speichersegment wird von OS/2 in periodischen Abständen auf die zuletzt durchgeführte Veränderung hin untersucht. Beim Auftreten einer Hauptspeicheranforderung (wenn der gesamte Hauptspeicher belegt ist) wird das Segment, das am längsten nicht mehr angesprochen wurde, auf Diskette-/Platte ausgelagert und der Speicherbereich freigegeben. Falls ein Programm später auf diesen Speicherbereich zugreift, tritt ein entsprechender Fehler auf. OS/2 fängt diesen Fehler ab und lädt die Information wieder von der Diskette/Platte in den Speicher, wobei dafür ein anderes Speichersegment ausgelagert wird, falls dies nötig sein sollte. Dieser Vorgang ist für die Anwendung nicht erkennbar; der einzige Nachteil besteht in einer kurzen Verzögerung durch das Laden des benötigten Speicherbereichs von der Diskette/Festplatte.

Die Konzepte der Speicherauslagerung und des virtuellen Speichers sind einfach, eine gute Implementierung dagegen nicht, denn OS/2 muß den richtigen Speicherbereich auf effiziente Weise auslagern. Dabei muß nicht nur darauf geachtet werden, daß die Auslagerungsdatei auf dem Datenträger nicht zu groß wird und dessen gesamte Kapazität belegt, sondern es darf auch keine Systemsperre eintreten. Sollte beispielsweise die Datenträ-

gerkapazität wegen einer übergroßen Auslagerungsdatei erschöpft sein, kann aus dem Hauptspeicher kein Speichersegment mehr ausgelagert werden. Nun ist kein freier Hauptspeicher verfügbar, in den Teile der Auslagerungsdatei wieder geladen werden können. Das System "hängt sich auf". Hinzu kommt, daß der Ablauf der Systemprogramme bei umfangreichen Speicherauslagerungen verlangsamt wird; die Verringerung der Ausführungsgeschwindigkeit muß für den Benutzer akzeptabel bleiben, und das System darf nicht in einer unlösbaren Situation (*deadlock*) hängen bleiben.

Die Aufgaben der Speicherverwaltung in OS/2 sind natürlich wesentlich komplizierter, als in diesem Überblick gezeigt werden konnte. Wir werden uns mit dem Gebiet der Speicherverwaltung in Kapitel 9 noch detaillierter befassen.

2.1.4 Schutz

Über die Speicherverwaltung wurde schon gesagt, daß von einem Anwendungsprogramm nicht unbedingt ein korrektes Verhalten erwartet werden kann. Dies trifft ebenso für den Systemschutz zu. OS/2 muß Anwendungen vor den Auswirkungen anderer ablaufender Applikationen auf dem gleichen System schützen.

Da Anwendungen und Programme von einer Vielzahl verschiedener Hersteller unter OS/2 ablaufen werden, sind fehlerhafte Systemzustände nur schwerlich exakt zu definieren; die Simulation auftretender Probleme ist praktisch unmöglich. Kein Softwareentwickler kann ein Produkt in allen denkbaren Umgebungen und Situationen austesten. Ein Programmfehler darf also nicht zum Systemabsturz oder zur Beendigung anderer Programme führen - oder noch schlimmer: zur Zerstörung oder Veränderung von fremden Daten, wobei das System weiterarbeitet und der Fehler nicht bemerkt wird. Nur wenige Benutzer besitzen genügend Kenntnisse über die Geräteausstattung zum Diagnostizieren eines Problems, das beim Ablauf einer Anwendung in einer Multitasking-Umgebung auftritt.

Hinzu kommt, daß Microsofts Zukunftsvision eines vollständig automatisierten Bürobetriebes nur mit einem geschützten System realisiert werden kann. Kein Unternehmen wird mit einem System arbeiten, bei dem jeder Benutzer in Personal- und andere sensitive Daten (Lohn-, Gehaltsabrechnungen oder ähnlich wichtige Informationen) Einblick erhalten kann.

Heute werden Personal Computer meist noch wie hochentwikkelte Taschenrechner verwendet. Die Daten werden mit traditionellen Mitteln gesichert (beispielsweise durch abgeschlossene Büroräume). Der Benutzer sieht keine Notwendigkeit für eine durch Software geschützte Umgebung, da das Gerät selbst physikalisch vor nichtautorisierten Zugriffen geschützt ist. Dieses fehlende Interesse bezüglich eines Software-Schutzes für Programme wie Daten auf einem PC liegt in einer ohne auf Datenschutz und Systemsicherheit zu achtenden Verbreitung des PCs in den letztem Jahren. Ein Systemschutz wird nicht benötigt (so meint der durchschnittliche Systembenutzer), da der Aufbewahrungsort selbst dem System eine gewisse Sicherheit bietet und die verarbeiteten Daten über traditionelle Kanäle eingebracht werden. In der Zukunft aber werden vernetzte Personal Computer eine immer größere Verbreitung finden und sowohl zur Bearbeitung als auch als Quelle (über das Netzwerk) von Daten eingesetzt werden. Von diesem Blickwinkel betrachtet stellt der Datenschutz eine Hauptanforderung dar und ist für Personal Computer in dieser zentralen Rolle unbedingt erforderlich.

2.1.5 Isolierung (Encapsulation)

Ein in einem Singletasking-System (wie es MS-DOS Version 3.X darstellt) ablaufendes Programm befindet sich immer in derselben gleichartigen Systemsoftwareumgebung, die sich aus dem Gerät und dem Betriebssystem zusammensetzt. Das Programm kann erwarten, daß es vom System immer gleichartig behandelt wird und die Interaktion mit dem Benutzer wohldefiniert ist. In einer Multitasking-Umgebung können mehrere Programme gleichzeitig ausgeführt werden. Jedes Programm benutzt dabei Dateien und Einheiten auf unterschiedliche Arten; ein Programm kann die Maus ansprechen, jedes Programm kann

die Bildschirmdarstellung in einem unterschiedlichen Modus erstellen usw. OS/2 muß jedes Programm "vereinzeln", vom Rest isolieren, so daß für eine Anwendung immer eine einheitliche Umgebung vorliegt.

2.1.6 Kommunikation zwischen Prozessen - interprocess communication (IPC)

In einer Singletasking-Umgebung wie unter MS-DOS Version 3.X wird der Mikroprozessor des PCs nur von einem Programm genutzt. Falls das Programm eine Dienstleistung benötigt, die nicht vom Betriebssystem bereitgestellt wird, muß es auf eine eigene Routine zurückgreifen. Ein Beispiel hierfür ist die Möglichkeit zur Durchführung eines Sortiervorgangs, die Sortierroutine muß vom Anwendungsprogrammen selbst zur Verfügung gestellt werden.

Dies trifft auch dann zu, wenn in eine Tabelle eines Tabellenkalkulationsprogramms Daten aus einer Datenbank eingelesen werden sollen. In diesem Fall muß im Tabellenkalkulationsprogramm der entsprechende Programmcode für das Einlesen der Datenbankdaten vorliegen. Dadurch wird im Tabellenkalkulationsprogramm eine besondere Funktion angeboten, das Programm vergrößert sich und wird auf ein Datenbankdateiformat festgelegt. Ein Benutzer kann ein leistungsfähigeres Datenbankprodukt vielleicht nicht einsetzen, da dessen Daten nicht in die Kalkulationstabelle übernommen werden können. Damit unterliegt die Anwendung des Tabellenkalkulationsprogrammes Einschränkungen.

Ein Ergebnis zur Lösung solcher Probleme, die im Bereich der Datenübertragung zwischen verschiedenen Anwendungen auftreten können, ist die Entwicklung eines großen und umfassenden All-Round-Paketes wie *Symphony*. Da jede vom Benutzer gewünschte Funktion in einem Programm vorhanden sein muß, entwickeln manche Hersteller Programme, mit denen alle Problemstellungen gleichermaßen gut zu lösen sind.

In der Praxis hat sich jedoch herausgestellt, daß solche Programme zu umfangreich und umständlich zu bedienen sind. Au-

ßerdem sind die einzelnen funktionellen Komponenten (Tabellenkalkulation, Textverarbeitung, Datenbank usw.) weit weniger gut entwickelt als die spezialisierter Einzelanwendungen (wie beispielsweise *Multiplan*).

Eine Singletasking-Umgebung erzwingt so die Entwicklung umfangreicher und komplexer Programme, wovon jedes mit eigenen Dateiformaten arbeitet und Daten (sofern überhaupt) nur sehr begrenzt übertragen kann. Die Vision einer PC-Software, die aufgrund ihres Leistungsumfangs nicht mehr realisierbar und für einen Benutzer wegen der umfangreichen Funktionsabläufe und Vorgänge nicht mehr beherrschbar ist, ergab eine weitere Forderung, die vom Betriebssystem OS/2 erfüllt werden mußte. Anwendungen müssen einfach und effizient mit anderen Anwendungen kommunizieren - einen Datenaustausch gewährleisten - können.

Das bedeutet, daß vom Betriebssystem eine Funktion zur Verfügung gestellt werden muß, damit eine Anwendung, die bestimmte Informationen oder Dienstleistungen bereitstellt, zu finden und zu benennen ist. Es muß eine effiziente Kommunikationsmöglichkeit für den Datenaustausch zwischen Anwendungsprogrammen geben, ohne daß man als Anwender mit diesen Abläufen belastet wird. So muß ein Tabellenkalkulationsprogramm mit einem Datenbankprogramm kommunizieren und auf benötigte Daten problemlos zugreifen können. Ein Tabellenkalkulationsprogramm kann auf diese Weise mit allen möglichen Datenbankprogrammen verwendet werden, die OS/2-IPC-Anforderungen bearbeiten können.

Die Anwendungen unter OS/2 beziehen ihre Leistungsstärke nicht nur aus den Vorteilen der Funktionalität und Effizienz einer Anwendung, sondern auch aus der Kommunikation aus einem Anwendungsprogramm heraus mit anderer auf dem PC unter OS/2 lauffähiger Software. Wird beispielsweise eine neue und leistungsfähigere Datenbank angeschafft, so wird damit auch indirekt die Leistungsfähigkeit der Tabellenkalkulation gesteigert, wenn diese auf die Dienstleistungen des Datenbankprogramms zurückgreifen kann.

Die OS/2-Philosophie ist also die: Kein Programm soll das Rad nochmals erfinden. Jedes der Programme, das die IPC-Anforderungen unter OS/2 erfüllt, kann so entwickelt werden, daß die Serviceroutinen von anderen Programmen genutzt werden können. Das Ergebnis ist dann eine maximale Effektivität und Effizienz für sämtliche Software-Pakete auf einem Computer.

2.1.7 Direkter Zugriff auf Einheiten

Die graphische Schnittstelle und die Begrenzung traditioneller Einheitentreiberarchitekturen wurden bereits angesprochen. OS/2 bietet eine Lösung für den Zugriff auf die Bildschirmschnittstelle. Andere spezialisierte Einheiten erfordern möglicherweise eine weitergehendere Schnittstellenspezifikationen, die mit den vorhandenen Einheitentreibern nicht realisiert werden können. Da zum heutigen Zeitpunkt keine Aussagen über die zukünftige Entwicklung von Geräteeinheiten und Peripheriegeräten möglich sind, hat man unter OS/2 zu diesem Zweck eine Ausweichmöglichkeit implementiert.

OS/2 bietet den direkten Zugriff auf Hardwareeinheiten (Ein-/Ausgabeports und Speichereinheiten). Dies wird realisiert, indem auf fest vorgegebene Art und Weise auf Geräteeinheiten zugegriffen wird. Anwendungen sollten den Zugriff vom Betriebssystem oder über einen Einheitentreiber ausführen lassen. Dies bietet den Vorteil, daß nichtstandardisierter Hardware wie optischen Zeichenerkennungsgeräten, Digitalisierungstabletts, FAX-Anschlüssen, speziellen Graphikadaptern usw. ohne Probleme ihre speziellen Einsatzfähigkeiten erhalten bleiben.

Spezialsoftware erfordert für die Anwendung unter einem Betriebssystem sicherlich spezialisierte Routinen. Diese können nicht zur Verfügung gestellt werden, wohl aber Konventionen, nach denen eine Implementation unter Einhaltung vorgegebener Regeln und Schnittstellenprotokolle angeboten werden kann.

2.2 Kompatibilitätserwägungen

OS/2 mußte aber noch weitere Forderungen erfüllen: Es muß sowohl mit der Architektur der Prozessoren 8086/8088 und 80286 kompatibel sein als auch mit früheren Versionen des MS-DOS. Die einfachste Lösung hätte in der Entwicklung eines völlig neuen Multitasking-Betriebssystems bestanden, das mit MS-DOS nicht kompatibel gewesen wäre. Ein solches System ist aber für den PC-Markt nicht akzeptabel, da es eine Unmenge von Anwendungen unter MS-DOS gibt. Potentielle OS/2-Benutzer würden das neue System nicht kaufen, erstens, solange keine Anwendungen dafür verfügbar sind und zweitens, wenn keine Ablauffähigkeit ihrer alten Anwendungen gewährleistet ist. Die Entwickler von Anwendungen würden aber keine Produkte entwickeln, solange sie keinen für sich lukrativen Markt erkennen können. In diesem Teufelskreis - keine Applikationen, da noch kein Markt vorhanden; kein Markt, da keine Anwendungsprogramme - hätte ein neues Betriebssystem keine Chance.

2.2.1 Real-Modus gegen Protect-Modus

Die ersten Kompatibilitätserwägungen bezüglich des *Real*-Modus entstanden beim Design des Mikroprozessors 80286 - der CPU eines MS-DOS-fähigen Computers. Der Mikroprozessor 80286 besitzt zwei inkompatible Modi - den *Real*- und *Protect*-Modus. Der *Real*-Modus wurde zur Emulation der Prozessoren 8086/8088 implementiert. Bei der Benutzung des 80286 im *Real*-Modus verhält sich der Prozessor wie ein 8088, nur ist der Durchsatz größer.

Der 80286 sollte aber mehr als ein schneller 8088 sein. Leistungsmerkmale wie die Speicherplatzverwaltung, den Speicherbelegungsschutz und Schutzmechanismen bei der Verwendung unterschiedlicher Ringe (Operationsmodi), die das Betriebssystem vor fehlerhaften Anwendungen schützen, sollten implementiert werden. Dabei konnte mit dem älteren Baustein 8088 und dessen Leistungsmerkmalen keine vollständige Kompatibilität erhalten bleiben. Man mußte über das Leistungsspektrum des 8088 hinausgehen. Aus diesem Grunde wurde ein zweiter Mo-

dus des 80286 hinzugefügt, der *Protect*-Modus. Beim Betreiben des Prozessors im *Protect*-Modus werden wichtige neue Leistungsmerkmale unterstützt; leider funktionieren die meisten Programme, die für den 8086/8088 entwickelt wurden, in diesem Modus aber nicht.

Der 80286 leistet also eigentlich die Arbeit von zwei Mikroprozessoren. Er kann sich wie ein schneller 8088 verhalten - kompatibel, aber ohne neue Leistungsmerkmale - oder er verhält sich wie ein 80286 - inkompatibel zum 8086, aber mit neuen Leistungsmerkmalen. Leider erkannten die Designer dieses Bausteins nicht die Bedeutung der Kompatibilität mit dem MS-DOS-Markt und entwickelten den 80286 daher so, daß er in jedem Modus funktioniert, aber nicht beliebig umgeschaltet werden kann. Der 80286 wurde so entwickelt, daß er *nur* die alten 8086/8088-Programme oder *nur* die Ausführung neuer 80286-Programme unterstützt.

Hinweis: Der 80286 initialisiert sich selbst im Real-Modus. Zum Umschalten in den Protect-Modus gibt es einen Befehl; die Rückumschaltung ist aber nicht mehr möglich.

Von OS/2 wurde nun erwartet, daß diese Problemstellung, die vom 80286 nicht gelöst wurde, vom Betriebssystem übernommen wurde. Es sollten gleichzeitig sowohl Programme im *Real*-Modus (8086/8088) als auch im *Protect*-Modus (80286) ausgeführt werden können.

2.2.2 Ablauf von Anwendungen im Real-Modus (Kompatibilitätsmodus)

Die Problemlösung - es konnte zwischen zwei Modi hin- und hergeschaltet werden - führte zu zusätzlichen Schwierigkeiten für die Systementwickler von OS/2. Diese beruhten in erster Linie darauf, daß die Real-Programme unter den MS-DOS Versionen 2 und 3 für eine Singletasking-Umgebung geschrieben wurden.

Beim MS-DOS wird normalerweise von einem "Betriebssystem" gesprochen; eigentlich ist es aber nur ein Hilfsmittel zur Ausführung diverser Anwendungsprogramme. In einer ungeschütz-

ten Umgebung können beliebige Anwendungsprogramme die Interruptvektoren und Peripherieeinheiten manipulieren und MS-DOS in nahezu jeder beliebigen Form verwenden. Diese Flexibilität des Betriebssystems ist einer der Gründe für den Erfolg von MS-DOS. Falls MS-DOS eine bestimmte Dienstleistung nicht bietet, kann man eine eigene Routine, ohne Konflikte zwischen DOS und Applikation, hinzufügen. Die Softwareentwickler konnten immer wieder neue Möglichkeiten austesten (häufig mit großem Erfolg). Für die meisten Anwendungen ist MS-DOS nur ein Programmlader mit einer Anzahl von Dateiverwaltungsroutinen. Zudem wird von MS-DOS eine Schnittstelle zur Hardware geboten (hier sind zum Beispiel die Abfrage von Interruptvektoren und das Editieren von Parametern in der Diskettenbelegungstabelle (FAT, *file allocation table*) möglich).

Im Normalfall sollte man vermuten, daß Anwendungssoftware, die spezialisierten Routinen für einen funktionsfähigen Ablauf neben den DOS-Routinen zur Verfügung stellen, unter einer neuen DOS-Version, wie es für OS/2 zutrifft, nicht ablauffähig ist. Aber tatsächlich ist das Gegenteil der Fall. Meist haben sich Anwendungsprogramme als Standard durchgesetzt, auf den ein neues Betriebssystem oder eine Weiterentwicklung Rücksicht nehmen muß. So müssen die Systementwickler bestehende Konventionen in ein neues System einfließen lassen; der "Applikationsstandard" muß unterstützt werden. Eine standardisierte Betriebssystemschnittstelle ist meist leider nicht bekannt. Das Entwicklungsteam einer DOS-Version ist so gezwungen, mit viel Experimenten zu arbeiten, um nichtdokumentierte Nebeneffekte und Verknüpfungen zu erkennen.

Es gibt drei Kompatibilitätsebenen. Die Verfügbarkeit einer MS-DOS-kompatiblen Anwendungsprogrammschnittstelle (API, *applications program interface*) ist die Kompatibilität der ersten Ebene. Falls Anwendungen weiterhin die Systemhardware manipulieren dürfen, so ist dies die Kompatibilität der zweiten Ebene. Die dritte Kompatibilitätsebene setzt eine Umgebung voraus, in der die unterschiedlichen Manipulationen eines Programms (für eine Singletasking-Umgebung) berücksichtigt werden.

2.2.2.1 Speicherausnutzung

Anwendungen im *Real*-Modus, für die eine Unterstützung von OS/2 erwartet wird, wurden tatsächlich für eine Singletasking-Umgebung entwickelt. Daher beanspruchen diese Programme den gesamten verfügbaren Systemspeicher (üblicherweise sind keine anderen Programme vorhanden, die auch Speicheranforderungen haben können). Selbst wenn ein Programm nicht den gesamten verfügbaren Speicherbereich anspricht, können Nachforderungen auftreten. Eine solche zusätzliche Speicheranforderung kann unter MS-DOS Versionen 2.X und 3.X nahezu immer ausgeführt werden. Auch von OS/2 muß eine solche Anfrage korrekt bearbeitet werden können, um eine volle Abwärtskompatibilität zu erhalten.

Schon die Art der Speicheranforderung weist auf ein Singletasking-System hin. Programme fordern den freien Speicher üblicherweise in zwei Schritten an. Zuerst wird nach dem maximalen Speicherbereich gefragt, den der Mikroprozessor - meist der 8088 - bereitstellen kann (das ist ein Megabyte). Jeder Anwendungsprogrammierer weiß, daß die Anforderung verweigert werden muß, da MS-DOS maximal 640 Kbyte Hauptspeicherkapazität verwalten kann. Bei der Anforderungsverweigerung wird der Anwendung mitgeteilt, wieviel Speicher im System vorhanden ist. Das Programm fordert dann genau diese Speichermenge an, und da dem Programmierer bekannt ist, daß die zweite Speicheranforderung nicht verweigert werden kann, so sind üblicherweise keine Sicherheitsabfragen in die Programme eingebaut worden. Ein Fehler wegen ungenügender Speicherkapazität kann bei einem zweiten Aufruf nicht mehr auftreten.

In einer Multitasking-Umgebung, wie OS/2 sie darstellt, führt dieses Verhalten zu Problemen. Nach der ersten Speicheranfrage wird von OS/2 die in diesem Moment verfügbare freie Speichermenge gemeldet. Da gleichzeitig mehrere Programme ausgeführt werden können, kann ein Teil der angegebenen Speicherkapazität bis zur zweiten Speicherforderung bereits anderweitig vergeben sein. In diesem Fall ist die zweite Speicheranforderung des ersten Programms immer zu hoch und führt zu einem Feh-

ler. Das Antworten von OS/2 mit einem Fehlercode hat wenig Sinn, da die *Real*-Modus-Anwendung nicht mehr auf einen Fehler prüft (sie wurde in dem Glauben entwickelt, daß das Auftreten eines Fehlers bei der zweiten Speicheranforderung unmöglich ist - Singletasking!). Dieses Verhalten führt dazu, daß OS/2 die zweite Speicheranforderung ebenfalls verweigert und die Anwendung einen ihr nicht zugewiesenen Speicherbereich belegt (in der Annahme, ihr sei der angeforderte Speicher zugewiesen worden). Die Folge ist die Zerstörung der Daten anderer Prozesse.

Offensichtlich war, daß OS/2 diese Problemstellung lösen und gleichzeitig die existierende Basis von Anwendungen im *Real*-Modus unterstützen mußte.

2.2.2.2 Dateisperre

Da in einem Multitasking-System zwei oder mehr Programme ablaufen können, ist ein gleichzeitiger Zugriff zweier Programme auf die gleiche Datei möglich. Beispielsweise versucht ein Programm, in eine Datei zu schreiben, während ein anderes Programm dieselben Daten liest. Multitasking-Systeme lösen diese Problemstellung üblicherweise durch einen Mechanismus, der *Dateisperre* genannt wird. Dabei wird jeweils nur einem Programm der Zugriff auf eine Datei erlaubt.

OS/2 liefert einen Rückgabecode für eine Dateisperre, wenn eine Anwendung den Zugriff auf die Datei verlangt, die wegen des Zugriffs eines anderen Programms gerade gesperrt ist. Das Anwendungsprogramm gibt den Zugriffsversuch üblicherweise auf oder wartet und versucht die Ausführung der Operation zu einem späteren Zeitpunkt erneut. Der OS/2-Rückgabecode für eine gesperrte Datei kann an eine Real-Modus-Anwendung nicht übergeben werden, da diese den Code nicht erkennt (unter MS-DOS Versionen 2.X und 3.X gab es diese Fehlerart nicht). Nur wenige Real-Modus-Anwendungen prüfen Lese- und Schreiboperationen auf Fehlercodes, und selbst wenn dies der Fall ist, verstehen sie den in der Antwort vom Betriebssystem gelieferten Code nicht.

Da Real-Modus-Programme im allgemeinen keine Möglichkeit haben, mit dieser Problemstellung fertig zu werden, muß OS/2 für die Programme entsprechende Aktionen übernehmen.

2.2.2.3 Netzwerk-Überlegungen

Beim Betrieb unter MS-DOS Version 3.1 kann eine Anwendung auf den *Network Virtual Circuit* eines existierenden Netzwerkes zugreifen, um mit einer Anwendung auf dem Server zu kommunizieren. Ein einziger Circuit kann für unterschiedliche Zwecke verwendet werden, da MS-DOS Version 3.1 nur eine Singletasking-Version ist. Der Netzwerk-Redirector verwendet den Circuit nur, wenn die Anwendung MS-DOS zur Ausführung einer Netzwerkfunktion veranlaßt. Da die CPU auf der Seite des MS-DOS liegt, kann keine Anwendungssoftware ausgeführt werden, die private Meldungen sendet (dadurch verbleibt der Circuit für den Redirector verfügbar).

Im Gegensatz dazu kann eine Anwendung, die eigene Meldungen sendet, MS-DOS nicht gleichzeitig ausführen und über den Redirector-Code (der in MS-DOS eingebaut ist) den Virtual Circuit benutzen.

In OS/2 ist dies nicht länger der Fall. OS/2 ist ein Multitasking-Betriebssystem und eine Anwendung kann folglich den Redirector zu einem Zeitpunkt ansprechen, zu dem die Real-Modus-Anwendung den Circuit benutzt. OS/2 muß den Zugriff auf die Circuits unterbinden, damit die Benutzeranforderungen in einem Netzwerk nicht miteinander in Konflikt geraten.

2.2.2.4 Kompatibilität häufig benutzter Funktionen

Als Ziele der Kompatibilität unter OS/2 für MS-DOS-Anwendungsprogramme wurde bisher erläutert, daß Anwendungen, die unter MS-DOS ablauffähig waren (eventuell mit internen Softwareschnittstellen), auf jeden Fall auch unter OS/2 benutzt werden können. Dies ist unter dem neuen Betriebssystem verwirklicht worden, da es wichtig war, daß existierende Anwendungen auch unter OS/2 korrekt funktionieren. OS/2 muß auch eine funktionelle Kompatibilität bieten; im Protect-Modus müs-

sen Anwendungen entwickelt werden können, für die die Funktionen des Real-Modus ebenfalls verfügbar sind.

Hinweis: Ein extrem hoher Kompatibilitätsgrad ist erforderlich, um Anwendungen mit dokumentierten und undokumentierten Schnittstellen und Leistungsmerkmalen unter OS/2 ablaufen zu lassen. Wenn eine dieser Schnittstellen nicht unterstützt wird, kann die Anwendung nicht korrekt funktionieren.

Dies ist schwierig, da viele bekannte Anwendungen (Routinen wie *SideKick*, die beendet werden, aber resident im Speicher verbleiben) für ungeschützte Singletasking-Umgebungen entwickelt wurden. Dabei wurde der Tatsache keine Beachtung geschenkt, daß die Funktionen dieser Programme in geschützten Umgebungen wesentlich einfacher implementiert werden können. So sind einige der Anwendungsprogramme nur einsatzbereit und so leistungsfähig, indem sie direkt in die Betriebssystemsoftware eingreifen, also durch Veränderung von MS-DOS selbst. Unter OS/2 kann ein Eingriff in das Betriebssystem nicht erlaubt werden (siehe Kapitel 4), so daß OS/2 eine alternative Methode im Protect-Modus bereitstellen muß, damit die vom Benutzer gewünschten Services auch hier geboten werden können.

2.2.3 Abwärtskompatibilität

Bis hierher war die Darstellung für die Kompatibilität OS/2-DOS lediglich auf die Aufwärtskompatibilität beschränkt. Alte Programme sollen unter dem neuen Betriebssystem laufen. Dies war aber nicht unbedingt umgekehrt ebenso.

Die Abwärtskompatibilität - die Fähigkeit von Programmen, die unter OS/2 entwickelt worden sind, auch unter MS-DOS ablaufen zu können - ist ebenfalls wichtig. Anwender entwickeln nur sehr zögernd OS/2-Anwendungen, solange das neue Betriebssystem noch keine große Marktdurchdringung erlebt hat. Andererseits wirkt sich die Nichtverfügbarkeit von Software auf die Marktdurchdringung negativ aus. Es ist daher möglich, Anwendungen zu entwickeln, die Leistungsmerkmale des Protect-Modus von OS/2 unterstützen und gleichzeitig ohne Veränderungen unter MS-DOS Version 3.X ausgeführt werden können. Auf

diese Weise werden OS/2-Produkte entwickelt, die auch heute schon unter MS-DOS 3.X vermarktet werden können.

2.2.3.1 Family API

Um die Abwärtskompatibilität für alle Anwendungen, die unter OS/2 entwickelt wurden, sicherzustellen, integrierten die Designer von OS/2 eine sogenannte *Family API* (*Family Application Program Interface* - damit ist die Verwandtschaft zwischen DOS und OS/2 angesprochen) für OS/2. Diese Schnittstelle bietet eine Standardumgebung sowohl unter MS-DOS Version 3.X als auch für OS/2. Die einheitliche Schnittstelle ermöglicht die Programmentwicklung für beide Betriebssystemwelten - MS-DOS und OS/2. Durch die Verwendung der Family API kann ein Programmierer eine Anwendung erstellen, die OS/2-Funktionen verwendet (aber auch Funktionen unter MS-DOS Version 3.X) und die als Binärprogramm sowohl unter MS-DOS Version 3.X als auch unter OS/2 ablauffähig ist. So können mit Hilfe der Family API-Schnittstelle OS/2-Funktionen in eine MS-DOS-Version-3.X-Umgebung integriert werden.

2.2.3.2 Netzwerkkompatibilität: Server-Benutzer

Eine andere wichtige Form der Auf- und Abwärtskompatibilität muß für Applikationen angeboten werden, die in Netzwerken betrieben werden. Heute kann man von einem OS/2-System in einem Netzwerk ausgehen, das nur mit MS-DOS-Systemen 3.1 kommuniziert. Eines Tages aber werden neue Versionen von OS/2 mit anderen OS/2-Versionen kommunizieren. Eine Netzwerkschnittstelle für OS/2 muß also sowohl aufwärts- als auch abwärtskompatibel für alle bereits bekannten und zukünftigen netzwerkfähigen MS-DOS-Versionen sein.

Kapitel 3

Strukturprinzipien für OS/2

Für das Softwaredesign ist eine Anzahl von Programmiertechniken maßgeblich geworden. Eine bestimmte Programmstruktur wird wegen des Codeumfangs, der Ablaufgeschwindigkeit etc. oder auf der Basis der Anweisungsstruktur (gut oder schlecht, je nachdem, ob es die Zielsetzungen trifft oder nicht) bewertet. Zweck und Konsistenz unterliegen der Design-Idee - der Programmentwurfsphilosophie; deren Einfluß wird in jeder einzelnen Beurteilung deutlich.

Der Zweck des Softwareentwurfs ist die Spezifikation von Richtlinien, nach denen sich die Systemanalytiker zur Lösung einer Aufgabe richten können. Für ein Projekt der Größe und des Umfangs von OS/2 mußte eine klare und allgemein verständliche Entwurfsgrundlage für alle Programmteile festgelegt werden, da das heutige OS/2 auch auf zukünftige Versionen dieses Betriebssystems Einfluß nimmt. Diese Strukturprinzipien werden auch aus einem zweiten Grund benötigt: zur Sicherstellung der Konsistenz für alle Programmierer, die im Entwicklungsteam mitarbeiten. Diese Durchgängigkeit der Strukturprinzipien der Programmentwurfsphilosophie ist von großer Wichtigkeit; falls ein Programmierer einen Programmteil (Aufgabe A) entsprechend einer vorgegebenen Philosophie optimiert, demgegenüber aber einen zweiten Teil (Aufgabe B) vernachlässigt, ein zweiter Programmierer aber das Gegenteil tut (also beide unterschiedliche Optimierungsstrategien verfolgen), so ist das Resultat, daß weder Programmierteil A noch B optimal ablaufen.

In diesem Kapitel werden die grundlegenden Entwurfsphilosophien für die Entwicklung von OS/2 dargelegt: maximale Flexibilität der Routinen, eine stabile und immer wieder verwendbare Umgebung, Optimierung der Fehlersuche und die Bereitstellung von Software-Tools für Softwareentwickler, um optimierte Anwendungsprogramme einfacher erstellen zu können.

3.1 Maximale Flexibilität

In der Einleitung dieses Buches wurde die Entwicklungsarbeit an OS/2 als ein Prozeß beschrieben, der die Grundlage für einen technischen Durchbruch in der Systemprogrammierung darstellte. Eine der einfachsten Voraussagen ist die, daß die genaue Entwicklung und Richtung der Neuerungen nicht abgeschätzt werden kann. Eine ähnliche Situation bestand damals beim Übergang der Prozessorgeneration vom 8080 auf den 8088 - von CP/M zu MS-DOS. So wurde damals der Prozessor 8088 mit der Fähigkeit zur Verwaltung von maximal einem Mbyte Hauptspeicher ausgerüstet. Computer der Gattung IBM PC und Kompatible besitzen aber nur einen adressierbaren Speicherbereich von 640 Kbyte. Zum Zeitpunkt dieser Entscheidung in der Systemprogrammierung (wieviel Speicherkapazität benötigt wird) waren 640 Kbyte Hauptspeicherkapazität zehnmal mehr als die damaligen 8080-Systeme ansprechen konnten; die ersten PCs wurden mit 16 Kbyte Hauptspeicherkapazität ausgerüstet und ein Hauptspeicherbereich von 640 Kbyte erschien damals mehr als genug. Trotzdem dauerte es nur wenige Jahre, bis diese Speicherkapazität als gering und heute als nahezu nicht mehr tragbar eingestuft wurde. Die Hardwareentwicklung hat die zu diesem Zeitpunkt gesetzten Grundlagen der Systementwicklung von MS-DOS rasch eingeholt - ja, sogar überholt.

Die Zukunft von OS/2 wurde offengehalten, indem ein möglichst flexibles System geschaffen wurde, in das viele zukünftige Leistungsmerkmale im Nachhinein implementiert werden können, sollte dies nötig sein. Auf diese Weise können neue OS/2-Versionen einfach freigegeben werden, die auf zukünftige Bedürfnisse ausgerichtet sind. Die heutigen Komponenten muß-

ten daher so entwickelt werden, daß dadurch sich ändernde Leistungsmerkmale in der Zukunft nicht behindert werden. Neue Versionen des MS-DOS müssen ebenfalls unterstützt werden.

Durch diese mehr oder weniger offene Entwurfsvorgabe waren die wichtigsten Prinzipien schon festgelegt. Alle Leistungsmerkmale sind auf die Unterstützung des Wandels von DOS zu OS/2 ausgerichtet, der unausweichlich ist:

1. Alle OS/2-Merkmale sollten einfach gehalten werden, so daß sie in zukünftigen Systemen, die sich vom Design unterscheiden, problemlos weiterhin unterstützt werden können. Entweder sind die Leistungsmerkmale selbst einfach oder sie wurden in Basisleistungsmerkmale eingebaut, die ihrerseits einfach gehalten wurden. Das Adjektiv *einfach* bezieht sich in diesem Zusammenhang nicht auf das externe Erscheinungsbild (eine geringe Anzahl von Funktionen und Optionen), sondern auf die interne Struktur des Betriebssystems zur Ausführung einer Funktion. Diese soll möglichst unkompliziert sein, so daß sie in zukünftigen Versionen nicht hinderlich wird.

   Zukünftige Versionen des Betriebssystems werden die älteren Versionen mit einem minimalen Aufwand unterstützen (Aufwärtskompatibilität). Die Entwicklung einheitlicher Betriebssystemversionen bedeutet mehr als in die Vergangenheit zu schauen; es bedeutet die Vorausschau in die Zukunft.

2. Alle Systemschnittstellen sollten eine Erweiterung in zukünftigen Versionen unterstützen. Bei der Abfrage des Status einer Diskettendatei muß dem Betriebssystem nicht nur ein Zeiger auf eine Struktur mit den nötigen Informationen übergeben werden, sondern die aufrufende Anwendung muß auch die Länge der Struktur angeben. Obwohl die aktuelle Version des Betriebssystems genau N Bytes übergibt, unterstützt eine zukünftige Version vielleicht andere Arten von Diskettendateien und liefert M Bytes an Informationen. Da die Anwendung dem Betriebssystem (über den Parame-

ter der Pufferlänge) mitteilt, welche Informationsstruktur verstanden werden kann (die alte, kurze Version oder die neue, lange Version) kann das Betriebssystem beide Arten unterstützen.

Im allgemeinen sollen Systemschnittstellen die aktuellen Leistungsmerkmale unterstützen, ohne neue Leistungsmerkmale zukünftiger Versionen zu unterbinden. In den Zähl- und Flaggenargumenten sollte für zukünftige Erweiterungen genügend Raum vorhanden sein und alle übergebenen Strukturen sollten ein Größenargument enthalten oder ihren Umfang selbst festlegen können (*self sizing*).

Eine weitere Schnittstelle verdient besondere Beachtung - die Schnittstelle zum Dateiverwaltungssystem. Eine Zusatzfunktion für das Dateiverwaltungssystem (zum Beispiel längere Dateinamen) anzubieten, ist nicht einfach, da viele ältere Anwendungen beispielsweise Dateinamen mit mehr als acht Zeichen nicht verarbeiten können und diese dann als ungültig zurückweisen. OS/2 löst solche und ähnliche Probleme durch die Festlegung, daß alle Dateinamen als mit Null beendete Strings (ASCIIZ-Strings) von willkürlicher Länge behandelt werden.

Programmierer sollten einen Dateinamen nicht zerlegen, anpassen oder gar ändern. Das empfangende Programm sollte einen Pfadnamen so entgegennehmen wie er angeboten wird. Ein Dateiname sollte an das Betriebssystem übergeben oder von dort erhalten, aber nicht vom aktuellen Programm bearbeitet werden. Die Schnittstellendetails werden in späteren Kapiteln ausführlich erläutert.

3. Den Betriebssystemfunktionen von OS/2 müssen jederzeit Funktionen hinzugefügt werden können. Die Implementationsdetails neuer Funktionen sollten verborgen bleiben. Ist diese Voraussetzung erfüllt, dann können diese Funktionen jederzeit verändert werden. OS/2 sollte für Anwendungen die Quelle eines neuen Leistungsmerkmals nicht offen legen. Einige APIs werden vom *Kernel*-Code bedient, andere von Unterroutinenbibliotheken und wieder

andere von im System ablaufenden Prozessen. Da eine Anwendung keinen Unterschied in der Verwendung von Betriebssystemroutinen machen kann, haben die Systemdesigner in den Änderungsanforderungen einer API-Implementation freie Hand. So kann beispielsweise eine OS/2-Kernel-API in zukünftigen Versionen verändert und die alte API durch Unterbibliotheksroutinen weiterhin unterstützt werden. Die Bibliotheksroutine würde die alte Form der API übernehmen, sie in die neue Form umwandeln, den OS/2-Kernel aufrufen und das Resultat dem umgekehrten Umwandlungsprozeß unterziehen. Dieser Ablauf erlaubt zukünftigen OS/2-Versionen die Unterstützung neuer Leistungsmerkmale im Betriebssystem, ohne daß alte Leistungsmerkmale für bereits existierende Programmsysteme nicht mehr unterstützt werden (siehe Kapitel 7).

4. Letzter Gesichtspunkt ist das Erreichen einer möglichst optimalen Flexibilität des Betriebssystems auch auf der niedrigsten Ebene. Mit anderen Worten, ein Benutzer sollte dem System Funktion hinzufügen können - zum Beispiel eine Datenbankverwaltung - oder gar eine Systemkomponente ersetzen können (beispielsweise ein neuer Graphiktreiber), ohne auf eine neue Version von Microsoft warten zu müssen. Ein Mikrocomputer-Design, das die Verwendung von Hardware und die Aktualisierung von Versionen durch Dritthersteller erlaubt, wird ein *offenes System* genannt. Alle IBM PCs und Kompatible sind ein klassisches Beispiel für offene Systeme. Ein Systementwurf, der solche Erweiterungen nicht ermöglicht, wird demgegenüber als *geschlossenes System* bezeichnet. Die erste Version des Apple Macintosh ist hierfür ein Beispiel. Auf den ersten Blick erscheint MS-DOS als geschlossenes Softwaresystem, da keine direkten Erweiterungsmöglichkeiten vorgesehen sind. In der Praxis betrachtet ist MS-DOS aber ohne jeden Zweifel ein offenes System, da es durch seine ungeschützte Umgebung jeder Anwendung erlaubt, auch im Betriebssystem selbst Veränderungen vorzunehmen und die Hardware direkt anzusprechen. Ein offenes Softwaresystem ist ebenso

wichtig wie ein transparentes und dokumentiertes Hardwaresystem. Da OS/2 selbst ein geschütztes Betriebssystem ist, wurden Leistungsmerkmale (wie das dynamische Linken) hinzugefügt, um Systemerweiterungen durch Microsoft, andere Softwarehersteller und natürlich auch durch den Benutzer selbst, ohne große Programmentwicklungsschwierigkeiten (siehe Kapitel 7) zu ermöglichen.

3.2 Eine stabile Umgebung

Ein Computersystem im Büroeinsatz muß seinen Benutzern eine stabile Umgebung zur Verfügung stellen. Jedes Anwendungsprogramm sollte bei der Ausführung immer wieder gleichartig ablaufen und mit einheitlichen Daten die gleichen Ergebnisse erbringen. Der Ablauf einer Anwendung sollte auf keinen Fall durch eine andere Anwendung beeinflußt werden können. Selbst beim Auftreten eines Programmfehlers sollte dieser keinen Einfluß auf andere, gleichzeitig ablaufende Anwendungen haben. Fehler sollten vom Betriebssystem erkannt, abgefangen und dem Benutzer mitgeteilt werden. Diese Forderung nach dem störungsfreien Ablauf von Anwendungsprogrammen ist sicherlich leicht verständlich, läßt sich jedoch auf den heute angebotenen PC-Systemen nicht verwirklichen.

3.2.1 Speicherschutz

Moderne Computer basieren auf der *von-Neumann-Architektur*; einer Architektur, die von John von Neumann, einem in Ungarn geborenen amerikanischen Mathematiker und Computerwissenschaftler erfunden wurde. Ein von-Neumann-Computer besteht aus nur zwei Teilen: einer Speichereinheit und einer Verarbeitungseinheit. Die Speichereinheit enthält sowohl die zu verarbeitenden Daten als auch die Anweisungen (das Programm) zur Steuerung der Verarbeitungseinheit. Die Verarbeitungseinheit liest die Instruktionen aus dem Speicher und diese Befehle können beispielsweise das Lesen weiterer Daten, die Verarbeitung bereits geladener Daten oder das Speichern von Daten veranlassen.

Ein von-Neumann-Computer unterscheidet nicht zwischen Anweisungen und Daten; beide werden als binäre Zahlen im Speicher des Computers abgelegt. Die einzelnen Programme müssen dafür sorgen, daß in bestimmten Speicherstellen Daten und in anderen Speicherstellen Instruktionen enthalten sind. Jedes Programm verwendet den Speicher dabei auf eine ihm eigene Art. Da der Computer nicht zwischen Daten und Instruktionen unterscheidet, kann ein Programm seine eigenen Daten als Instruktionen verarbeiten.

Hinweis: Dies ist selbstverständlich eine grobe Vereinfachung. OS/2 und die 80286-CPU enthalten Leistungsmerkmale, die zwischen Instruktionen und Daten unterscheiden und die Fähigkeit eines Programms zur Modifikation eigener Instruktionen beschränken (siehe 9.1).

Sehen wir uns an, welche Verarbeitungsschritte OS/2 bei der Ausführung eines Programms zu erledigen hat: Das Programm wird in den Speicher eingelesen (und die Instruktionen dabei wie Daten behandelt), und die Daten in den einzelnen Speicherstellen werden anschließend ausgeführt. So ist es einem Programm sogar möglich, Instruktionen zu manipulieren und sich selbst zu reprogrammieren.

Programme sind meist komplex, und logische Fehler können zu unerwünschten Änderungen von Daten und Instruktionen im Speicher führen. So könnte ein Programm einen Befehlspuffer mit einer Länge von 80 Bytes anlegen und vom Benutzer erwarten, daß dieser keine Befehle eingibt, die die vorgegebene Länge überschreiten. Falls ein Benutzer einen längeren Befehl eingibt und das Programm diese Eingabe nicht überprüft, werden unter Umständen dadurch die hinter dem Befehlspuffer abgelegten Daten überschrieben, und das System kann abstürzen.

Hinweis: Dies ist ebenfalls ein einfaches Beispiel. Aktuelle Programme prüfen die einfachen Fehlermöglichkeiten meist ab; in einer etwas komplexeren Form treten solche Fehler aber durchaus auf.

In einer Singletasking-Umgebung - MS-DOS Version 3.X - läuft zu einem Zeitpunkt immer nur eine einzige Anwendung ab. Ein im Anwendungsprogramm auftretender Fehler kann den Speicherbereich von MS-DOS, der Anwendung oder den bisher vom Programm unbenutzten Speicher verändern. In der Praxis (wegen der Konventionen zur Speicherverwaltung) wird der

Speicherbereich von MS-DOS nur selten unvorhergesehenen Änderungen unterliegen. Meist zerstört ein fehlerhaftes Programm eigene oder beliebige andere Daten in einem unbenutzten Speicherbereich. Ein solcher Fehler wird unter Umständen nicht sofort bemerkt, vor allem die Fehlerursache läßt sich nicht ermitteln. In jedem Fall erzeugt das Programm inkorrekte Ergebnisse oder das System stürzt ab. In den letzten beiden Fällen verliert der Benutzer die Arbeit seit der letzten Speicherung, es wird aber immer klar, in welcher Anwendung der Fehler liegt (die Anwendung, die zum Zeitpunkt des Systemabsturzes ausgeführt wurde). Zur Vervollständigung sei hinzugefügt, daß es vorkommen kann, daß Anwendungen den Speicherbereich von MS-DOS weitgehend unbemerkt verändern, so daß die fehlerhafte Anwendung noch scheinbar korrekt beendet wird. Bei der Ausführung der nächsten Anwendung ist das System dann aber meist nicht mehr in der Lage, seine Aufgabe ordnungsgemäß durchzuführen. Häufig stürzt der Computer bereits beim Laden des nächsten Programmes ab.

Es ist offensichtlich, daß Fehler in Programmen, die in einer Singletasking-Umgebung ablaufen, relativ einfach gefunden werden können. Das bedeutet nicht, daß die Fehlersuche bei Programmen in Singletasking-Systemen eine einfache Aufgabe ist. Ganz im Gegenteil, es ist nur eindeutig, in welchem Programm der Fehler liegt; nicht klar hingegen ist, an welcher Stelle in der Anwendung der Fehler auftritt.

Wenn mit der fehlerhaften Anwendung gleichzeitig ein zweites Programm ausgeführt wird, wird die Situation noch komplizierter.

Der am häufigsten auftretende Fehler in Anwendungsprogrammen liegt in der Speicheradressierung. Es werden dann Speicherbereiche angesprochen, die von MS-DOS für das Programm nicht belegt wurden. In einer Singletasking-Umgebung sind diese Speicherstellen üblicherweise unbenutzt, in einer Multitasking-Umgebung aber sind sie oftmals anderen Programmen zugeordnet. Das Programm wird inkorrekte Ergebnisse liefern, Speicherbereiche zerstören, Daten überschreiben, das System

zum Absturz bringen oder zu einer Kombination dieser Möglichkeiten führen. Ein Speicheradressierungsfehler ist in einem Multitasking-System gefährlich, da hier auch Speicherbereiche (und damit Daten oder Programmcode) anderer Programme zerstört werden können und ein Datenverlust verhängnisvolle Auswirkungen haben kann.

Weitere Schwierigkeiten können durch Anwendungsprogramme dann auftreten, wenn sie nicht miteinander arbeiten können. Dies sind keine Programmfehler, sondern Fehler im Programmablauf; ein aktives Programm kollidiert mit einem zweiten Programm. Ein einfaches Beispiel ist das *Verbiegen des Tastaturvektors* (siehe Bild 3.1).

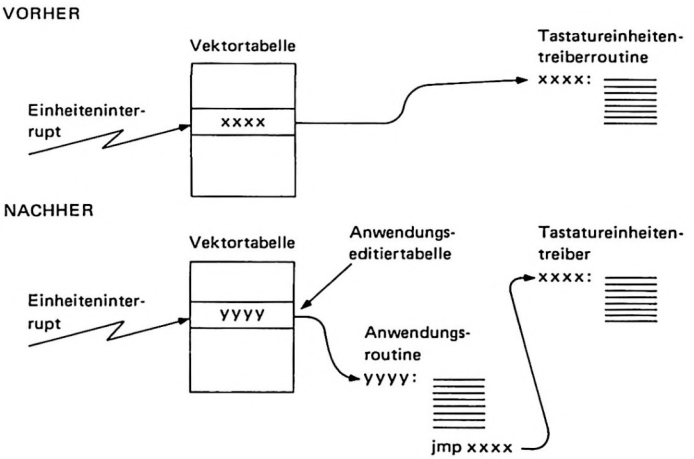


Bild 3.1 Verbiegen des Tastaturvektors

Eine Anwendung verändert den Inhalt einer Speicherstelle von MS-DOS, so daß beim Drücken einer Taste ein vom Anwender geschriebenes Programm (statt des MS-DOS-Codes) ausgeführt wird. Diese Methode dient der Untersuchung und Abfrage be-

stimmter Ereignisse (wie der Betätigung der Umschalttaste ohne gleichzeitige Betätigung einer anderen Taste). MS-DOS übergibt dann einen solchen Tastendruck nicht an die Anwendung - die Tastaturabfrage wird abgefangen. Bei der Ausführung nur einer einzigen Anwendung ist das Verbiegen des Tastaturvektors sicherlich empfehlenswert, obwohl dabei Speicherstellen verändert werden, die der Anwendung nicht zugeordnet sind. Üblicherweise ist ein solches Vorgehen in einer Singletasking-Umgebung erfolgreich. In einer Multitasking-Umgebung dagegen kann eine zweite Anwendung den gleichen "Trick" versuchen. Ergebnis ist dann jedoch, daß das Gesamtsystem sicherlich nicht mehr korrekt arbeiten wird.

Damit wird deutlich, daß eine stabile Umgebung den Schutz von Speicherbereichen notwendig macht. Ein Speicherbereich, der einer Anwendung nicht zugeordnet ist, darf weder bewußt noch unbewußt von einer anderen Anwendung verändert werden.

3.2.2 Schutz vor Nebeneffekten

Eine stabile Umgebung macht mehr als nur den Schutz von Speicherbereichen nötig; sie erfordert, daß die Ausführung von Programmen keine Nebeneffekte für andere ablaufende Anwendungen mit sich bringen. Dabei spielt es keine Rolle, ob ein Nebeneffekt schwerwiegende Auswirkungen hat oder nicht, in jedem Fall wird ein *Grundsatz einer stabilen Umgebung* damit untergraben.

Betrachten wir als Beispiel noch einmal das Verbiegen des Tastaturinterruptvektors. Hier wird durch eine Anwendung die Tastatureingabe abgefangen. Dies trifft in der Multitasking-Umgebung aber nicht nur für die eine gewünschte Applikation zu, sondern gilt auch für andere Anwendungen. Der Nebeneffekt in diesem Fall ist katastrophal - die den Vektor beeinflussende Anwendung erhält auch alle Tastatureingaben, die nicht für sie bestimmt waren und andere Anwendungen warten auf Tastenanschläge, die an dieser Stelle niemals ankommen.

Nebeneffekte im Programmablauf können auch bei der Verwendung vom Hersteller freigegebener Systeme auftreten, wenn die einzelnen Leistungsmerkmale, Komponenten und Kommunikationsebenen im Programm nicht sorgfältig geplant und ausgetestet wurden. So unterstützt beispielsweise das Großrechnerbetriebssystem TOPS-10 Befehlsdateien, die ähnlich den MS-DOS-Stapelverarbeitungsdateien aufgebaut sind. Gleichzeitig ist in diesem Betriebssystem aber auch die Befehlsausführung im Offline-Betrieb möglich. Da beide Programme dasselbe TOPS-10-Leistungsmerkmal für den Offline- und Stapelverarbeitungsbetrieb ansprechen, kommt es zur gegenseitigen Beeinflussung, wenn eine Stapelverarbeitungsdatei im Offline-Betrieb verarbeitet wird.

OS/2 behandelt die Problemstellung "Nebeneffekte" durch eine möglichst umfangreiche Virtualisierung der Betriebssystemumgebung für jede Anwendung. Das heißt, daß OS/2-Anwendungen in einer Standardumgebung eingebunden werden, die von Umgebungsänderungen anderer Anwendungen nicht beeinflußt werden kann. Vergleichbar ist dies mit gleichartigen Wohnungen in einem Mehrfamilienhaus. Jeder Mieter zieht ein und kann seine eigene Wohnung (sprich seine Umgebung) beliebig einrichten, die Wohnung des Nachbarn wird aber von der Einrichtung des ersten Mieters nicht beeinflußt.

Im folgenden finden Sie einige Beispiele zu Anwendungsumgebungen, die OS/2 virtualisiert.

- *Arbeitsverzeichnisse.* Jede Anwendung besitzt ein Arbeitsverzeichnis (aktuelles Verzeichnis) für jedes angeschlossene Laufwerk. Wenn beispielsweise unter MS-DOS Version 3.X ein Kindprozeß das aktuelle Verzeichnis für Laufwerk C verändert und beendet wird, so ist das Arbeitsverzeichnis für Laufwerk C auch für den Ursprungsprozeß verändert. OS/2 eliminiert diesen Nebeneffekt durch eine separate Liste der Arbeitsverzeichnisse für jeden Prozeß im System. Wenn eine Anwendung das Arbeitsverzeichnis verändert, bleibt die Festlegung des Arbeitsverzeichnisses für andere Anwendungen unverändert.

- *Verwendung von Speicher.* Die Speicherzuweisung kann ebenfalls Nebeneffekte verursachen. Wenn ein Prozeß den gesamten verfügbaren Hauptspeicher für sich beansprucht, verbleibt kein Speicherbereich für andere Anwendungen. Das Speicherverwaltungssystem des OS/2 verwendet die Speicherauslagerung (*swapping*), so daß den Speicheranfragen der einzelnen Anwendungen nachgegangen werden kann.

- *Prozeßpriorität.* OS/2 verwendet einen auf der Basis von Prioritäten beruhenden Scheduler zur Zuweisung der CPU-Zeit an einzelne Prozesse. Applikationen können die eigene und die Priorität ihrer Kindprozesse beliebig festlegen. Aber auch die Prioritätsfestlegung eines Tasks kann Nebeneffekte hervorrufen. Stellen Sie sich einen Prozeß unter OS/2 vor, der mit einer höheren Priorität als jeder andere Prozeß im System ablaufen muß. Falls ein zweiter Prozeß dieselbe Anforderung stellt, tritt ein Konflikt auf: Beide Prozesse können nicht die höchste Priorität im System besitzen. Im allgemeinen hängt die Priorität eines Prozesses von den Prioritäten anderer Prozesse im System ab. Der OS/2-Scheduler enthält einen hochentwickelten Absolut-/Relativ-Mechanismus zur Behandlung solcher Konflikte.

- *Verwendung von Dateien.* Durch Anwendungsprogramme werden Dateiinhalte verändert, die unter anderem auch von einer zweiten Anwendung bearbeitet werden. Auch hier können Probleme und Nebeneffekte auftreten. Unerwünschte Dateimodifikationen werden durch eine Dateisperre, die unter OS/2 implementiert ist, ermöglicht. Die gleichzeitige Aktualisierung einer Datei durch zwei Anwendungsprogramme wird durch die Sperrung einzelner Datensätze ermöglicht.

- *Umgebungsstrings.* OS/2 übernimmt das MS-DOS-Konzept der Umgebungsstrings. Im Gegensatz zu MS-DOS erhält OS/2 aber nicht nur einen einzigen Satz von Umgebungsstrings für das gesamte System; unter OS/2 verfügt jeder Prozeß über eigene Umgebungsstrings. Falls beispielsweise

ein Kindprozeß eine Kopie der Umgebungsstrings des Ursprungsprozesses erbt, so führen Veränderungen in den Zeichenketten nicht zur Änderung der Originalstrings in der Umgebung des Ursprungsprozesses.

- *Tastaturmodi.* OS/2-Applikationen können die Tastatur in zwei Modi betreiben - im *Filter-* und *Nichtfiltermodus* (*raw modus*). Je nach gewähltem Modus behandelt das Betriebssystem OS/2 oder die Anwendung die Anschläge von Steuertasten(-kombinationen). Ein solcher Aufruf wirkt sich auch auf nachfolgende Leseoperationen von der Tastatur aus; daher verfügt OS/2 über einen Datensatz, der den Status (Filter- oder Nichtfiltermodus) für jede Anwendung im System enthält. Falls eine bestimmte Anwendung mit der Tastatur arbeitet, wird in den jeweils entsprechenden Modus umgeschaltet.

3.3 Fehlersuche

Ein Schlüsselelement für eine stabile Umgebung ist die Systemunterstützung bei der Fehlersuche. Fehler werden immer gemacht und so muß davon ausgegangen werden, daß in allen Programmen Fehler vorliegen. Obwohl OS/2 Fehler nicht verhindern kann, bietet es doch leistungsfähige Möglichkeiten zur Lokalisierung und Suche von Fehlern. Die Fehlersuche besteht aus zwei Aktivitäten: Minimierung der Fehlerauswirkungen auf andere im System aktive Anwendungen und Maximierung der Benutzermöglichkeiten zur Bestimmung, welche Anwendung einen Fehler verursacht hat. Diese beiden Systemhilfsleistungen sind miteinander verknüpft, da ein Betriebssystem, das Fehler gegenüber der Umgebung abschirmt und vereinzelt, auch klarer erkennen läßt, wo der Fehler aufgetreten ist.

Der wichtigste Aspekt bei der Lokalisierung von Fehlern wurde bereits ausführlich erläutert - die Speicherverwaltung und deren Schutz. Die Prinzipien zur Fehlersuche sind folgende:

- Kein Programm kann das System abstürzen lassen. Ein wesentliches Element der OS/2-Entwurfsphilosophie war, daß kein Anwendungsprogramm das System (bewußt oder unwissentlich) zerstören kann. Sollte dennoch einmal das Betriebssystem von einer Anwendung so beeinflußt werden, daß es zu einem Absturz kommt, dann hat einerseits das System einen Fehler nicht erkannt und andererseits ist der Benutzer nicht in der Lage, die Anwendung, die den Fehler verursacht hat, zu bestimmen (da das System keine Hinweise mehr auf das Fehlerverhalten geben kann).
- Kein Programm darf eine Bildschirmgruppe außer seiner eigenen beeinträchtigen. Wie im weiteren Verlauf des Buches zu sehen sein wird, können bereits in den Zielsetzungen und der Entwurfsphilosophie sich wiedersprechende Bestrebungen zu erkennen sein, die zu Konflikten führen. Dies ist besonders auffällig, wenn entgegen aller Designvorgaben pragmatische Lösungen für Probleme gefunden werden müssen. So widerspricht zum Beispiel das Prinzip der Unterdrückung von Nebeneffekten der Anforderung zur Unterstützung von Tastaturerweiterungsprogrammen. Der einzige Zweck einer solchen Erweiterung ist aber gerade die Verursachung von Nebeneffekten - hierzu zählt insbesondere die Umwandlung bestimmter Tastenanschläge oder -kombinationen in andere Tastenanschlagsequenzen. OS/2 löst den Konflikt durch die Überprüfung und Veränderung des Datenflusses von und zu den Geräteeinheiten durch ein eigenes und speziell dafür entwickeltes Programm. Auf diese Weise kann eine Tastaturmakroanwendung, die Tastenanschläge unterdrückt, zwar die eigene, dem Prozeß zugeordnete Bildschirmgruppe unbenutzbar machen, nicht aber auf die Möglichkeit des Benutzers zum Umschalten zwischen Bildschirmgruppen Einfluß nehmen.

Ein Tastaturmonitor eines Anwendungsprogramms kann die Zeichen oder die Zeichenfolge, die über die Tastatur eingegeben werden, bis auf die Tastenanschläge, die OS/2 zur Umschaltung zwischen Bildschirmgruppen benutzt (Ctrl-Esc und Alt-Esc), abfangen und verarbeiten. Diese Maßnahme dient dem Benutzerschutz. So kann ein Programm alle Tastatureingaben selbst verarbeiten und die notwendigen Aktionen ablaufen lassen. Jedoch verbleibt immer noch die Umschaltmöglichkeit von einem Prozeß zum anderen erhalten.

- Anwendungsprogramme können keine GP-Fehler (GP steht für *general protection*) abfangen. Ein GP-Fehler tritt auf, wenn ein Programm ungültige Speicherstellen oder gültige Speicherstellen in einer nicht erlaubten Weise anspricht (schreiben in den Nur-Lese-Speicher). OS/2 unterbindet solche Operationen und zeigt dem Benutzer eine entsprechende Meldung an. Ein GP-Fehler ist ein Beweis für eine inkorrekte Programmlogik. Meist kann nicht davon ausgegangen werden, daß dies dem Benutzer bekannt ist oder sich das Programm von selbst verbessert. Demnach bietet das Betriebssystem eine Hilfestellung an.

  OS/2 erlaubt die Bearbeitung nahezu jeden anderen Fehlers durch das Anwendungsprogramm. Tritt beispielsweise ein Fehler bei der Eingabe eines ungültigen Dateinamens auf, so ist diese Fehleingabe vom Benutzer vorgenommen worden, sie rührt nicht von der Anwendung her. Die Anwendung kann diese Fehler direkt beheben (durch Korrektur der Eingabe oder eine erneute Eingabeaufforderung). Treten Fehler auf, die darauf zurückzuführen sind, daß Diskettenlaufwerke nicht bereit sind, so werden diese üblicherweise durch OS/2 behandelt. Die Korrektur kann aber auch vom Anwendungsprogramm übernommen werden. Dies ist insbesondere für Programme interessant, die unbeaufsichtigt im Hintergrund ablaufen. Solche Programme müssen Fehlersituationen lösen können, ohne auf eine Benutzereingabe zu warten.

3.4 Software-Tools

Im zweiten Kapitel wurde besprochen, was unter IPC zu verstehen ist, und es wurden die Vorteile der Benutzung einzelner Funktionen in voneinander unabhängigen Programmen erläutert. Die Vorteile und Flexibilität einer solchen Lösung aus unterschiedlichen Programmen gegenüber einer integrierten Gesamtlösung in einem Softwarepaket wurden ebenfalls erläutert. Dabei wurde deutlich, daß die Funktionalität des Gesamtsystems durch die Aktualisierung einzelner Programme (Installation neuerer, leistungsfähigerer Anwendungsprogramme) gesteigert werden kann. Alle diese Überlegungen gelten für Anwendungsprogramme (Standardsoftware) und beziehen sich auf die sogenannten Software-Tools.

Das englische Wort *Tool* heißt zu deutsch *Werkzeug* und der Ausdruck *Software-Tools* kommt daher, daß die einzelnen Programme und Anwendungen als Werkzeuge angesehen werden: Jedes Element - jedes Werkzeug - erledigt seine Aufgabe und diese sehr gut. Ein Handwerker, der eine Schraube drehen und einen Nagel einschlagen möchte, wird einen Schraubendreher und einen Hammer verwenden, aber kein Handwerkzeug, das zwar beide Aufgaben erledigen kann, dies aber nicht optimal. Im übertragenen Sinne kann man dies auch auf die auf einem Computer lauffähige Standardsoftware beziehen: Mit einer Tabellenkalkulation kann man ausgezeichnet finanzwirtschaftliche Aufgaben bewältigen, mit einem Graphikprogramm sehr effektiv Schaubilder erstellen und mit einem Datenbankpaket seine Daten optimal verwalten. Jedes Einzelpaket erledigt die ihm eigene Aufgabe sicherlich besser als dies ein integriertes Paket tun kann.

Nun wird manchmal behauptet, daß diese Aufteilung in einzelne Komponenten - je ein Programm für eine Aufgabe - ineffizient ist. Man könnte so argumentieren, daß die Aufsplittung in Einzelpakete sowohl zeit- als auch kostenintensiv ist, da verschiedene Aufgaben (beispielsweise Speichern, Datei laden) doppelt angeboten werden müssen - pro Paket einmal. Dieses Argument ist insofern richtig, als daß der Hauptspeicher und

die CPU-Zeit durch die Verwendung von Programmschnittstellen (zwar geringfügig, aber) belastet wird. Der Zeit- und Performance-Gewinn durch die Übertragung einer Aufgabe an eine dafür spezialisierte Routine wird dabei allerdings außer Acht gelassen. Durch den geringfügigen Mehraufwand der Programmaufrufe bleibt dem Benutzer aber die Freiheit zum Softwareausbau.

Der Benutzer erzielt aus dem modularen Aufbau des Systems also einen Nutzen. Wird - nun übertragen wir die Situation auf die Hardware und deren Komponenten - eine höhere Diskettenkapazität im System benötigt, wird einfach eine neue Festplatte eingebaut. Dabei kann frei zwischen den Angeboten verschiedener Hersteller gewählt werden. Die geringfügigen Kosten der Verbindungsteile und Treiberbausteine fallen bei diesen Überlegungen nicht mehr ins Gewicht, wenn man die zusätzlichen Leistungsmerkmale und die hinzugewonnene Systemflexibilität betrachtet.

Die Modularität der Software ist sicherlich das Äquivalent zu dem offenen System der Hardware. Dies ist ein wichtiger Teil des OS/2-Systementwurfs: Obwohl das Betriebssystem selbst nicht diese Modularität für die Softwareanwendungen, die unter OS/2 eingesetzt werden sollen, beansprucht und erfordert, sollte diese Funktionsweise in jeder Hinsicht unterstützt und das Betriebssystem selbst auf diese Weise aufgebaut werden.

Teil 2
Die Architektur

Kapitel 4

Multitasking

Die Ziele und Kompatibilitätserwägungen, denen OS/2 genügen mußte, sowie die Entwurfsphilosophie, auf deren Grundlage OS/2 implementiert ist, wurde bisher beschrieben. In den folgenden Kapiteln werden einzelne Betriebssystemstrukturelemente ausführlich beschrieben, wobei nicht nur die Funktionsweise der einzelnen Bausteine, sondern auch ihre Rolle im Gesamtsystem berücksichtigt wird.

In einem Multitasking-Betriebssystem können zwei oder mehrere Programme gleichzeitig ausgeführt werden. Der Vorteil eines solchen Leistungsmerkmals ist offensichtlich: Der Benutzer kann zwischen verschiedenen Anwendungsprogrammen umschalten, ohne seine Arbeit vollständig abbrechen zu müssen und ohne alle Daten zu sichern und ein Programm vor dem Start eines anderen zu beenden. Wird beispielsweise telefonisch eine Auskunft verlangt, wenn sich ein PC-Benutzer gerade mit der Textverarbeitung beschäftigt, kann die Anwendung unterbrochen und ein Kalenderprogramm für eine Terminabsprache aufgerufen werden oder eine Tabellenkalkulation, um dem Anrufer die nötigen Informationen für eine Vertragsvergabe sofort zur Verfügung zu stellen.

Multitasking ähnelt dem Verhalten von Personen, die nicht mit einem Computer arbeiten. Ein Bericht bleibt, halb gelesen, auf dem Schreibtisch liegen, da eine dringendere Aufgabe in der Zwischenzeit erledigt werden muß. Nach der Erledigung der anderen Aufgaben ist dann wieder Zeit vorhanden, und es wird an dem Bericht weitergearbeitet. So kann der Berichtschreiber seine Arbeit auch kurzzeitig für die Beantwortung eines Telefonanrufs unterbrechen. Es ist nicht nötig, daß das Projekt be-

endet und die Materialien beiseite gelegt werden, um zwischenzeitlich einen Telefonanruf zu beantworten. Lediglich die Aufmerksamkeit des Benutzers richtet sich zeitweise auf eine andere Aufgabe.

Diese Art des Multitasking wird *serielles Multitasking* genannt, da eine Aktion sequentiell nach einer anderen ausgeführt wird. Da diese Arbeitsweise natürlich ist, bedarf es gar keiner Überlegungen, welche Struktur oder Philosophie dahinter steckt. Wir arbeiten automatisch so. Ein Beispiel: Das Lesen der Tageszeitung ist serielles Multitasking - viele Informationen strömen auf den Leser ein, der einen Artikel nach dem anderen liest; dabei vielleicht einen Artikel anliest, einen anderen interessanteren Artikel findet und diesen liest, bevor der erste Artikel weitergelesen wird. Obwohl das serielle Multitasking nützlich ist, gibt es ein noch leistungsfähigeres Merkmal, das *parallele Multitasking* - mit dieser Art des Multitasking arbeitet OS/2.

Das parallele Multitasking auf einem Computer erfordert mehr Aufmerksamkeit und Kontrollfunktionen als dies beim Ablauf eines einzelnen Prozesses zu einem Zeitpunkt nötig ist. Personen führen auch serielles Multitasking aus, indem sie beispielsweise Auto fahren und sich gleichzeitig unterhalten. Oder es wird gestrickt und gleichzeitig Radio gehört. Keine dieser Handlungen erfordert allerdings die vollständige Konzentration. Wir können das parallele Multitasking nur dann ohne große Einschränkungen durchführen, wenn die verschiedenen Aufgaben keine hohe Konzentration benötigen.

Aber auch der Personal Computer hat nur einen Mikroprozessor - eine CPU. OS/2 kann die Aufmerksamkeit der CPU mehrere hundert Male in der Sekunde auf unterschiedliche Anwendungsprogramme richten. Für menschliche Zeitbegriffe scheinen wegen der hohen Geschwindigkeit des Programmwechsels alle Programme gleichzeitig abzulaufen. Wenn beispielsweise fünf Programme zu einem Zeitpunkt ausgeführt werden und jedes der Anwendungsprogramme einen Zeitbereich von 0,01 Sekunden (10 Millisekunden) die CPU benutzt, erhält jedes Programm in einer Sekunde 20 *Zeitscheiben*. Für uns Beobachter scheinen

alle fünf Programme gleichzeitig abzulaufen, tatsächlich werden sie in einem Fünftel ihrer maximalen Ablaufgeschwindigkeit ausgeführt. Auf die Aufteilung der CPU-Zeit werden wir später noch zurückkommen; im folgenden soll erst einmal angenommen werden, daß alle Programme gleichzeitig ausgeführt werden können.

Hinweis: Obwohl Computer mit mehreren CPUs heute bereits verfügbar sind, ist dieses Design doch noch ungewöhnlich. Die Erläuterungen beziehen sich in jedem Fall auf Systeme, die mehrere Recheneinheiten umfassen.

Die parallele Multitasking-Leistungsfähigkeit von OS/2 macht den Personal Computer zu mehr als einem "Gerät" zur Ausführung von diversen Anwendungsprogrammen. Der Personal Computer wird damit zu einem Dienstleistungssystem. Der Benutzer kann mit einem Tabellenkalkulationsprogramm arbeiten, während eine Mail-Anwendung Netzwerkmeldungen empfängt, die später gelesen werden. Zur gleichen Zeit laden andere Programme Daten von einem Mainframe-Computer oder geben Daten im Spooler-Betrieb an einen Drucker oder Plotter aus. Der Benutzer kann mehrere der Aktivitäten explizit gestartet haben, weitere Prozesse können von anderen Programmen aufgerufen werden. Die Programme laufen alle gleichzeitig ab und erledigen ihre Aufgabe, ohne daß sie explizit die Beachtung des Benutzers erfordern.

Auch für Applikationsprogramme selbst ist dieses Multitasking nützlich. Die Vorteile des modularen Aufbaus der Software wurden bereits erläutert - Programme können dabei so entwikkelt werden, daß sie ihre Dienstleistungen anderen Programmen anbieten können, in die diese Teillösungen dann nicht mehr integriert sein müssen. Die vielfältigen Vorteile dieser Arbeitsweise werden aufgrund des parallelen Multitasking noch erweitert. Ruft beispielsweise ein Programm ein anderes Programm zum Sortieren einer Datendatei auf, muß das Sortierprogramm zum gleichen Zeitpunkt wie das aufrufende Programm ausgeführt werden. Es ist nicht sinnvoll, wenn das aufrufende Programm zur Ausführung des Sortierprogramms beendet und anschließend wieder gestartet werden muß.

Das parallele Multitasking ist auch beim Ablauf eines Programms selbst nützlich. Hierbei kommt es vor allem auf das gleichzeitige Abarbeiten einzelner Programmteile an. Um diese "Gleichzeitigkeit" zu ermöglichen, gibt es sogenannte *Threads*. Threads werden in diesem Buch noch detaillierter beschrieben. Wir wollen im Moment davon ausgehen, daß wir es mit einem Computersystem mit mehreren CPUs zu tun haben, die parallel das Anwendungsprogramm ablaufen lassen können. Diese sehr vereinfachte Sichtweise soll zunächst die Einsicht in komplexe Vorgänge erleichtern. Auf diese Weise kann ein Anwendungsprogramm mehrere Aufgaben zum selben Zeitpunkt ausführen. Wenn der Benutzer beispielsweise ein Tabellenkalkulationsprogramm eine umfangreiche Budgetanalyse neu berechnen läßt, kann das Programm einen Thread für die Berechnung verwenden und einen weiteren zur Abfrage des nächsten Benutzerbefehls. Mehrere Operationen überlappen sich so in ihrer Ausführung und verbessern das Antwortverhalten des Programms gegenüber dem Benutzer.

Zur Unterstützung der parallelen Multitasking-Fähigkeit verwendet OS/2 einen prioritätsgesteuerten Vorrang-Scheduler, der mit Zeiteinheiten (*time slices*) arbeitet. Der OS/2-Scheduler entzieht einer Anwendung die CPU-Rechenleistung zu einem vom Scheduler festgelegten Zeitpunkt und weist die Rechenkapazität einer anderen Anwendung zu. Also geben die Programme nicht die CPU frei, wenn sie dies für richtig halten, sondern OS/2 verteilt die Belegung der CPU für jedes Programm. Jedes Programm (genauer gesagt, jeder Thread) im System besitzt eine bestimmte Priorität. Tritt ein Thread mit höherer Priorität als der aktuell ausgeführte auf, lenkt der Scheduler die Aufmerksamkeit der CPU auf den Thread mit der höheren Priorität. Falls zwei auszuführende Threads mit der gleichen Priorität auftreten, verteilt OS/2 die CPU-Aufmerksamkeit jeweils für vorgegebene Zeiteinheiten - für den Bruchteil einer Sekunde - auf jeden Thread.

Der OS/2-Scheduler sucht nicht nach Threads, die eine höhere Priorität als der aktuell ausgeführte Thread besitzen. Ein solches Verhalten würde Rechenzeit verschwenden und die Antwortzei-

ten erhöhen, da ein Thread mit höherer Priorität warten muß, bis der nächste Suchlauf des Schedulers ausgeführt wird. Statt dessen ruft ein anderer Betriebssystembestandteil den Scheduler beim Auftreten eines Threads mit höherer Priorität auf.

4.1 Das Modell der Subtasks

Die Begriffe *Task* und *Prozeß* werden hier beide im gleichen Sinne zur Beschreibung der Ausführung einer Binärdatei (.EXE) verwendet. Ein Prozeß gilt als Einheit unter OS/2, die über Speicherkapazität, geöffnete Dateien, Verknüpfungen zu *Dynlink*-Bibliotheken und Flaggen beim Ablauf verfügt. Viele Benutzer würden einen *Prozeß* auch *Programm* nennen, und tatsächlich bestehen alle Anwendungen und Programme unter MS-DOS aus einem einzigen Prozeß. OS/2 verwendet die Ausdrücke *Task* oder *Prozeß*, da hier ein Anwendungsprogramm aus mehreren Prozessen bestehen kann.

Zur Erzeugung oder Ausführung eines Prozesses wird ein anderer Prozeß (der aufrufende Prozeß, der auch *Vaterprozeß* genannt wird) benötigt; der aufgerufene Prozeß heißt *Kindprozeß*. Der Vaterprozeß des aufrufenden Prozesses ist quasi der *Großvaterprozeß* des *Kindprozesses* und so weiter. Wie auch bei der Analogie unter Menschen benötigt jeder Prozeß im System einen *Vaterprozeß*. Die Ausnahme: Beim Start von OS/2 wird ein Prozeß erstellt, der keinen Vaterprozeß besitzt (dies kann für die folgenden Betrachtungen aber unberücksichtigt gelassen werden). Das zum Vergleich herangezogene Beispiel der Vater-Kind-Beziehung trifft ebenfalls nicht genau zu. Ein Kindprozeß unter OS/2 ähnelt mehr einem Angestellten in einem Unternehmen, dem eine bestimmte Aufgabe zugeteilt wurde, die auszuführen ist. Der Vaterprozeß (in diesem Fall der Vorgesetzte oder Unternehmer) stellt einen Arbeitsplatz und die Informationen, die der Angestellte zur Ausführung seiner Aufgabe benötigt. Für einen Kindprozeß gilt diese Analogie der Darstellung (Unternehmer/Angestellter) ebenfalls. Bei der Erzeugung eines Kindprozesses erbt der Kindprozeß einen Großteil der Umgebung des Vaterprozesses. Der Ausdruck *erben* ist ebenfalls nicht

ganz richtig, da dies einbezieht, daß der Vaterprozeß beendet ist (was ja nicht stimmt). Der Kindprozeß übernimmt Rahmenbedingungen, die vom Vaterprozeß gesetzt werden.

Der wichtigste Teil der Umgebung, die ein Kindprozeß erbt, sind die geöffneten Dateinummern. OS/2 verwendet einen *Nummernmechanismus* zur Ausführung der Dateiein-/-ausgabe (wie unter MS-DOS ab Version 2.0). Beim Öffnen einer Datei gibt OS/2 eine Nummer (einen Integerwert) an den Prozeß zurück. Wenn ein Programm nun aus einer Datei lesen oder in eine Datei schreiben möchte, wird die Dateinummer der anzusprechenden Datei an OS/2 übergeben. Dateinummern können nicht zwischen verschiedenen Prozessen ausgetauscht werden. So kann beispielsweise die Datei, auf die sich ein Prozeß mit der Nummer 6 bezieht, keine Verbindung zu einer anderen Datei mit der Dateinummer 6 eines anderen Prozesses besitzen, sofern die beiden nicht in einer Vater-Kind-Beziehung stehen. Bei der Anlage eines Kindprozesses übernimmt dieser die geöffneten Dateinummern des Vaterprozesses. Ist beispielsweise die Datei *\Arbeit\Tempdat* für einen Vaterprozeß mit der Dateinummer 5 geöffnet, so bezieht sich auch der Kindprozeß über die Dateinummer 5 auf diese Datei.

Falls Ihnen dieses Prinzip nicht vertraut ist, wird Sie die Komplexität und Umständlichkeit vielleicht verwirren. Was soll der Kindprozeß mit den geöffneten Dateien des Vaterprozesses beginnen? Was hindert den Kindprozeß vor der Zerstörung der Dateien des Vaterprozesses? Dies alles wird klarer, wenn ein weiterer Teil in diesem Puzzle eingefügt wird - die *Standarddateinummern*.

4.1.1 Standarddateinummern

Viele OS/2-Funktionen verwenden 16-bit-Ganzzahlwerte, genannt *Nummern*, zum Zugriff auf Schnittstellen, das sind Dateien, Einheiten und Geräte. Eine *Nummer* ist ein Objekt, auf das der Programmierer keinen Einfluß besitzt - es ist ein willkürlicher Wert, den OS/2 an eine Anwendung zurückgibt, damit diese den Wert in nachfolgenden Aufrufen verwenden kann. Der

Zweck liegt in der Vereinfachung der OS/2-Schnittstellen und der Erhöhung der Ausführungsgeschwindigkeit einzelner Routinen. Bei der Erstellung einer Systemsemaphore wird dem Programm eine *Semaphorennummer* zurückgegeben, die für nachfolgende Anforderungs- und Freigabeoperationen benutzt wird. Der Zugriff auf eine Semaphore über den 16-bit-Wert ist wesentlich schneller als beispielsweise die Übergabe eines Dateinamens. Hinzu kommt, daß die 16-bit-Nummer des Anwendungsprogramms keine Informationen über die angesprochene Einheit liefert. OS/2 hat diesen Wert erstellt und was sich hinter der Semaphore verbirgt, hat von nun an nur noch für OS/2 Bedeutung; die Anwendung arbeitet zukünftig mit dieser Nummer und ruft Einheiten wie Dateien oder Geräte nicht direkt auf.

Dateinummern (*file handle*) sind eine besondere Form der Nummern, da mit ihnen Einheiten adressierbar sind. Die Dateinummern haben sowohl für die Anwendung als auch für das Betriebssystem eine besondere Bedeutung. Insbesondere drei Dateinummern sind vordefiniert: 0 für STDIN (für *standard input*), 1 für STDOUT (für *standard output*) und der Wert 2 für STDERR (für *standard error*). Ein einfaches Programm (NUMADD) soll die Erläuterung der drei Standarddateinummern erleichtern. NUMADD liest zwei Zeilen eines ASCII-Textes (von denen jede eine Dezimalzahl enthält), konvertiert die Zahlen in das Binärzahlensystem, addiert sie und wandelt das Resultat wieder in eine ASCII-Zeichenkette um, die dann in die Ausgabeeinheit geschrieben wird. Beachten Sie, daß es sich hier um ein einfaches Programm (nicht bildschirmausgabeorientiert) handelt, das als Werkzeug (*Tool*) entweder direkt vom Programmierer oder von einem anderen Programm (siehe Bild 4.1 und Listing 4.1) eingesetzt werden kann.

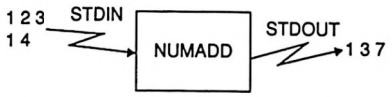


Bild 4.1 Das interaktive Programm NUMADD

```
#include <stdio.h>    /* definiert stdin und stdout */
main()
{
        int wert1, wert2, summe;

        fscanf(stdin, "%d", &wert1);
        fscanf(stdin, "%d", &wert2);
        summe = wert1 + wert2;
        fprintf(stdout, "%d\n", summe);
}
```

Listing 4.1 Programm NUMADD

Alle OS/2-Programme lesen ihre Daten aus der STDIN-Einheit und schreiben die Ausgabe nach STDOUT (dies ist festgelegt); alle Fehlermeldungen werden direkt nach STDERR geschrieben. Ein Programm wird die Einheiten nicht selbst öffnen, sondern übernimmt sie vom Vaterprozeß. Der Vaterprozeß kann die Dateien selbst geöffnet oder von seinem Vaterprozeß geerbt haben. Das Programm NUMADD enthält aus diesem Grund keine *DosOpen*-Aufrufe und beginnt über die Funktion *fscanf()* direkt mit dem Zugriff auf die Einheit mit der Dateinummer 0 (STDIN), die wiederum DOS-Leseaufrufe (*DosRead*) ausführt. Es wird die Funktion *fprintf* verwendet, welche über den Aufruf der Dateinummer 1 (STDOUT) DOS-Schreibaufrufe ausführt.

In Bild und Listing 4.2 wird die mögliche Anwendung NUMARITH verkürzt gezeigt. NUMARITH liest drei Textzeilen. Die erste Zeile enthält den Operator (zum Beispiel Plus oder Minus) und die zweite und dritte Zeile enthalten die Werte, mit denen die Berechnung durchgeführt werden soll. Mit dem Programm NUMARITH soll der Aufruf eines Unterprogramms (in diesem Fall NUMADD) dargestellt werden. Der Vaterprozeß NUMARITH besitzt eine Verbindung von STDIN zur Tastatur und die Standardausgabe ist mit dem Bildschirmeinheitentreiber verknüpft. CMD.EXE erbt die Dateinummern von seinem Vaterprozeß. Bei der Ausführung von NUMADD wird die Tastatureingabe über STDIN gelesen. Nachdem der Benutzer zwei Zah-

len eingegeben hat, zeigt NUMADD das Resultat über STDOUT auf dem Bildschirm an. NUMARITH hat das Programm NUMADD zur Ausführung einer Aufgabe aufgerufen. Der Aufruf, die Ausführung und die Datenübergabe von NUMADD ist für den Benutzer (oder ein aufrufendes Programm) nicht erkennbar. NUMADD hat als "Angestellter" von NUMARITH die Ein-/Ausgabeströme von NUMARITH verwendet; die Ausgabe von NUMADD wird, vom Benutzergesichtspunkt, NUMARITH zugerechnet.

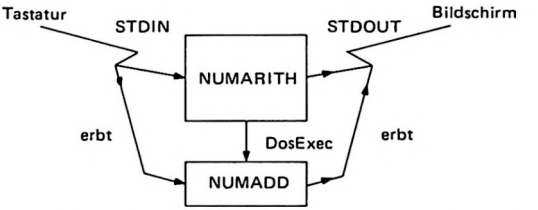


Bild 4.2 Interaktive Operation des Programms NUMARITH

```
#include <stdio.h>
  /**  numarith - Ausführung von Berechnungen
   *  Numarith liest Zeilentriplets:
   *
   *    operation
   *    wert1
   *    wert2
   *
   *    Die angegebene Operation (+, -, *, /) wird mit zwei Werten
   *    ausgeführt und das Ergebnis nach STDOUT geschrieben.
   **/
main()
{
        char operation;
        fscanf (stdin, "%c", &operation);
        switch (operation) {
                case '+': execl ("numadd", 0);
                        break;
```

```
                case '-': execl ("numsub", 0);
                        break;
                case '*': execl ("nummul", 0);
                        break;
                case '/': execl ("numdiv", 0);
                        break;
        }
}
```

Listing 4.2 Programm NUMARITH

Eine ähnliche Programmsituation ist in Bild 4.3 dargestellt. Die Einheiten STDIN und STDOUT von NUMARITH sind auf zwei Dateien gerichtet, aus denen Daten gelesen und in die Daten geschrieben werden sollen. Auch hier verrichtet NUMADD seine Aufgabe, ohne die Aufmerksamkeit des Benutzers auf sich zu lenken. Die Eingabewerte werden aus der mit STDIN verknüpften Datei gelesen und die Ausgabe in die mit STDOUT verknüpfte Journaldatei geschrieben. Der entscheidende Aspekt in dieser Programmkonstruktion ist, daß NUMADD das gleiche Programm ist, das auch von NUMARITH in Listing 4.2 aufgerufen wurde. NUMADD ist in beiden Fällen dasselbe Programm! In beiden Beispielen liest NUMADD aus STDIN und schreibt nach STDOUT; für NUMADD spielt dabei der Verknüpfungsprozeß Programm-Einheit über die Dateinummer keine Rolle. Auch für den Vaterprozeß - NUMARITH - ist dies gültig. NUMARITH kann nicht erkennen, ob mit Dateien oder der Tastatur und dem Bildschirm gearbeitet wird; es werden lediglich die STDIN und STDOUT-Dateinummern vom Vaterprozeß übernommen.

Das ist die Grundlage der Arbeitsweise von OS/2, die für eine Verknüpfung und dem Aufruf eines Prozesses aus einem anderen heraus eine Rolle spielt. Die Weitergabe der Rahmenbedingungen eines Vaterprozesses, die einem Kindprozeß vererbt werden, tragen dazu bei, daß der Vaterprozeß die Basisbedingungen für einen geregelten Ablauf setzt (siehe 4.1.2 Anonyme Pipes) - so zum Beispiel für das Lesen aus STDIN, von einer beliebigen Datei oder aus einem anderen Prozeß. Der Vaterpro-

zeß besitzt dadurch eine nahezu unbegrenzte Flexibilität in der Verwendung eines Tool-Programms und der Programmierer muß bei der Programmentwicklung nicht alle möglichen Situationen berücksichtigen.

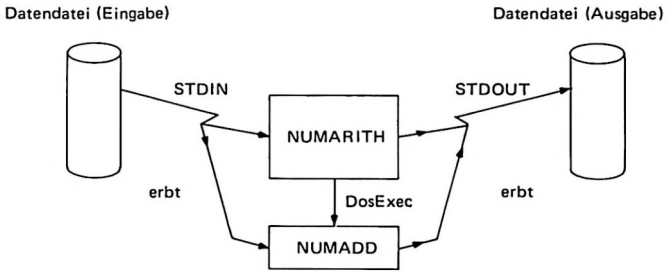


Bild 4.3 Operation des Programms NUMARITH mit Dateien

Von gleicher Wichtigkeit in der Vererbungsarchitektur ist die geschachtelte Verkapselung des Kindprozesses. Der NUMARITH aufrufende Vaterprozeß weiß nicht, wie NUMARITH seine Aufgabe erfüllt. NUMARITH könnte die Addition der beiden Zahlen sicherlich selbst durchführen oder NUMADD als Kindprozeß aufrufen. In jedem Fall wird die Ausführung der Addition nicht dem Vaterprozeß bekannt werden. Die Einbeziehung von NUMADD in die Ausführung von NUMARITH wird vor dem Vaterprozeß verborgen. Die Entscheidungen und Handlungen des Vaterprozesses von NUMARITH werden ebenfalls verborgen. NUMARITH hat keine Kenntnis von der verwendeten Datei oder Einheit, die durch den Vaterprozeß festgelegt wurde. Das Programm NUMARITH verfügt lediglich über die Dateinummern und führt die Verarbeitung der Programmanweisungen aus. Diese Arbeitsweise von Programmen ist sehr wichtig: NUMADD kann seinerseits einen Kindprozeß aufrufen, der einen Teil einer Aufgabe übernimmt, usw. Weder NUMARITH noch dessen Vaterprozeß wissen vom Aufruf des Unterprozesses. Andere Versionen des Programmsystems können jederzeit Teile der Anwendung ersetzen. Die neue Version kann mehr oder weniger Kindprozesse aufrufen oder in ihrer Aus-

führung verändert sein. Solange dem Vaterprozeß dasselbe Ergebnis geliefert wird, sehen Benutzer und Vaterprozeß keinen Unterschied. Die Systemarchitektur des Betriebssystems OS/2 basiert auf der Verwendung von Tools. Solange die Funktion eines Tools unverändert bleibt und das Ergebnis korrekt ist, bleibt die Implementation zweitrangig und kann beliebig verändert werden.

Die Standarddateinnummern STDIN, STDOUT und STDERR können von allen Programmen benutzt werden. Dies gilt auch dann, wenn keine Operationen mit Hilfe dieser Dateinummern durchgeführt werden (siehe Kapitel 14 Interaktive Programme).

4.1.2 Anonyme Pipes

NUMADD und NUMARITH sind einfache Programme; eine kurze Überlegung zeigt, wie die Vererbungsstruktur auch auf komplexere Programme anzuwenden ist. Ein Beipiel ist das Programm TREE, das beim Aufruf des Befehls TREE an CMD.EXE übergeben und ausgeführt wird. TREE erbt die STDIN- und STDOUT-Nummern, verwendet STDIN aber nicht. Die Ausgabe wird an STDOUT übergeben. Nachdem der Benutzer TREE hinter der CMD.EXE-Eingabeaufforderung eingegeben hat, erscheint die entsprechende Ausgabe auf dem Bildschirm. Beim Auftreten des Befehls TREE in einer Stapelverarbeitungsdatei erscheint die Ausgabe in der Journaldatei oder auf dem Bildschirm (abhängig davon, mit welcher Einheit STDOUT verknüpft ist).

Dieses Programmausführungsprinzip ist sehr nützlich; es kann allerdings die Situation auftreten, daß die Ausgabe eines Kindprozesses vom Vaterprozeß weiterverarbeitet werden soll. OS/2 bewältigt diese Aufgabe mit Hilfe der sogenannten *anonymen Pipes*. Mit dem Begriff *anonym* werden diese Pipes von den *benannten Pipes* unterschieden. Solche benannten Pipes sind in OS/2 Version 1.0 nicht implementiert.

Eine anonyme Pipe ist ein Datenpuffer, der unter der Kontrolle von OS/2 steht und vom Betriebssystem gewartet wird. Beim Öffnen einer anonymen Pipe durch einen Prozeß werden zwei

Dateinummern an den Prozeß übergeben - eine für das Schreiben und eine für das Lesen von Daten. Daten können unter Verwendung der "Schreibnummer" über einen *DosWrite*-Aufruf geschrieben und unter Verwendung der Lesenummer über einen *DosRead*-Aufruf gelesen werden. Eine anonyme Pipe ähnelt einer Datei, in die geschrieben oder aus der gelesen wird (über Dateinummernaufrufe); es gibt aber einige grundlegende und wichtige Unterschiede zwischen anonymen Pipes und Dateien. Einerseits werden die Daten einer Pipe in RAM-Puffern gespeichert (nicht auf einer Diskette) und andererseits kann die *DosSeek*-Operation auf Pipe-Nummern nicht angewandt werden.

Der Nutzen der Verwendung einer anonymen Pipe in einem einzelnen Prozeß ist gering, da die Pipe hier als FIFO-Puffer (*first in first out*) von limitierter Größe behandelt wird und die Daten mit *DosWrite*- und *DosRead*-Aufrufen in die Pipe-Puffer kopiert werden müssen. Was das Prinzip der anonymen Pipes so wertvoll macht, ist die Möglichkeit der Dateinummernweitergabe an den Kindprozeß. Ein Vaterprozeß kann eine anonyme Pipe anlegen und anschließend einen Kindprozeß initiieren; der Kindprozeß erbt die Dateinummern der anonymen Pipe. Mit der Schreibnummer kann der Kindprozeß in die Pipe schreiben und der Vaterprozeß kann die Daten über die Lesenummer entgegennehmen. Durch die Erweiterung mit der Funktion *DosDupHandle*, die die erneute Zuweisung einer Nummer an eine Einheit erlaubt, und den Standarddateinummern STDIN, STDOUT und STDERR steht ein leistungsfähiges System für flexible Ein- und Ausgabe zur Verfügung.

Doch zurück zum Beispiel mit den Programmen NUMARITH und NUMADD. Da NUMARITH die Dienstleistung des Programms NUMADD in Anspruch nehmen und die vom Additionsprogramm errechneten Ergebnisse weiter bearbeiten will, muß die Ausgabe dem Programm NUMARITH zugänglich sein und nicht nur auf dem Bildschirm abgebildet werden. Hinzu kommt, daß NUMADD nicht die Argumente aus der NUMARITH-Eingabedatei lesen soll; NUMARITH soll die Argu-

mente für NUMADD direkt bereitstellen. Dazu sind folgende Schritte notwendig:

1. Anlegen von zwei anonymen Pipes.

2. Sichern der alten, vor dem Programmablauf gültigen Konfiguration von STDIN und STDOUT (ob nun Datei, Einheit oder Pipe). Dies geschieht mit der Funktion *DosDupHandle*, mit der Dateinummern dupliziert werden können. Diese Duplikatnummern können beliebige Zahlen sein, solange sie nicht den Dateinummern von STDIN, STDOUT und STDERR entsprechen. Dies ist im Beispielfall sichergestellt, da *DosDupHandle* eine unbenutzte Nummer zuweist und die Standarddateinummern immer in Benutzung sind.

3. Über einen *DosClose*-Aufruf werden STDIN und STDOUT geschlossen. Dabei ist es gleichgültig, von welchen Programmen aus die Dateinummernaufrufe erfolgen, da die Einheiten für die Anwendung noch unter einer anderen Nummer geöffnet sind.

4. Nun wird *DosDupHandle* nochmals aufgerufen, um die Dateinummer von STDIN zu duplizieren und das Duplikat an die Pipe weiterzugeben, die die Zahl als Nummer der Eingabeeinheit erkennt. Das Duplikat der Dateinummer STDOUT wird zur Dateinummernausgabe der Pipe. Damit sind die Ein- und Ausgabe umgeleitet.

5. Initiieren Sie den Kindprozeß über einen *DosExecPgm*-Aufruf.

6. Schließlich werden die Dateinummern STDIN und STDOUT, die auf die Pipes zeigen, geschlossen und *DosDupHandle* und *DosClose* zur Umleitung auf die ursprünglich von STDIN und STDOUT angesprochenen Einheiten verwendet.

Das Ergebnis dieses Prozesses wird in Bild 4.4 gezeigt.

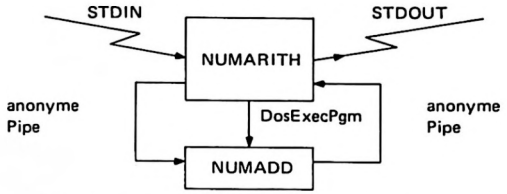


Bild 4.4 Aufruf von NUMADD über eine anonyme Pipe

Die STDIN- und STDOUT-Nummern von NUMADD zeigen auf zwei anonyme Pipes; der Vaterprozeß ist mit dem anderen Ende der Pipes verknüpft. Der Vaterprozeß verwendete *DosDupHandle* und *DosClose* zur temporären Umbenennung der STDIN- und STDOUT-Nummern. Der Kindprozeß erbt die Pipe-Nummern von STDIN und STDOUT des Vaterprozesses. An diesem Punkt kann der Vaterprozeß (NUMARITH) Eingabewerte in die Pipe schreiben, die mit der STDIN-Einheit von NUMADD verknüpft ist. Die Ausgabe von NUMADD kann aus der mit der STDOUT-Nummer verknüpften Pipe von NUMADD gelesen werden.

Ein Vergleich von Bild 4.4 mit dem Listing 4.2 und den Bildern 4.4 und 4.3 zeigt ein weiteres grundlegendes Merkmal des modularen Programmentwurfs: NUMADD besitzt keinen Programmcode oder spezielles Design zur direkten Kommunikation mit dem Programm NUMARITH. NUMADD funktioniert unabhängig von den verwendeten Geräteeinheiten auch mit anderen Programmen. Die Programmstruktur ist rekursiv: Wenn NUMADD einen Kindprozeß als Dienstleistungsroutine aufruft, können keine Kommunikationsprobleme zwischen den beiden Prozessen auftreten. Welchen Mechanismus NUMADD auch immer zur Kommunikation mit einem Kindprogramm nutzt: Es ist für das Programm NUMARITH völlig uninteressant und nicht nachzuvollziehen. NUMARITH kann als Tool für beliebige andere Programme verwendet werden, wobei das aufrufende Programm vom Ablauf der NUMARITH-Implementierung nicht betroffen ist; hier interessiert nur die Ergebnisübergabe.

Das Beispiel hat einen weiteren wichtigen Gesichtspunkt gezeigt. Im Normalfall wird ein Prozeß *DosRead*- und *DosWrite*-Aufrufe zur Ausführung der Ein- und Ausgabe über eine Pipe vornehmen; das Programm NUMADD verwendet aber *fscanf* und *fprintf* (zwei C-Bibliotheksroutinen). *fscanf* und *fprintf* führen *DosRead*- und *DosWrite*-Aufrufe aus, da die Nummer einer Pipe von einer Datei- oder Einheitennummer für die Operation nicht unterscheidbar ist. Nicht nur NUMADD funktioniert ohne Veränderung mit Pipes, sondern auch die davon aufgerufenen Bibliotheksunterroutinen. Dies ist ein weiteres Beispiel für das Prinzip der Vereinzelung, das in OS/2 realisiert ist. Unterschiede zwischen Pipes, Dateien und Einheiten werden voreinander verborgen und die Angabe von Nummern als standardisierte Nummernschnittstelle behandelt. Unter UNIX gilt dieser Aspekt gleichermaßen, diese Betriebssystemstruktur wurde von der UNIX-Architektur übernommen.

4.1.3 Weitere Details

Bei der bisherigen groben Darstellung der Task- und Tool-Architektur des Betriebssystems OS/2 wurden einige wichtige Details vernachlässigt. Diese werden nun erläutert.

STDERR (Dateinnummer 2) ist die Ausgabenummer, an die alle Fehlermeldungen übergeben werden. Die Fehlerausgabeeinheit STDERR wird benötigt, da STDOUT als normale Ausgabeeinheit für alle Programme eingesetzt wird. Bei der Eingabe folgender Zeile tritt ein Fehler auf, wenn die Einheit *datei* nicht existiert:

```
D datei > logdatei
```

Falls keine Standardfehlereinheit (STDERR) existiert, würde keine Fehlermeldung auftreten und statt dessen eine Journaldatei (*logdatei*) angelegt werden. Wollte der Benutzer später einmal die von ihm vermeintlich erstellte Datei aufrufen, würde das Computersystem folgende Meldung anzeigen:

```
FILE <datei> NOT FOUND
```

STDERR ist immer auf den Bildschirm gerichtet, und die Ausgabe wird von Anwendungen nur selten umgeleitet.

Einheiten, die eine spezielle Aufgabe übernehmen - STDIN, STDOUT und STDERR - sind nicht im OS/2-Kernel implementiert; sie basieren auf Systemkonventionen. Alle Programme sollten diesen Maßgaben folgen, um die Flexibilität und Nützlichkeit der auf Tools basierenden Architektur des OS/2 nicht einzuschränken. Selbst Programme, die keine Ein-/Ausgabe über die STD-Dateinummern ausführen, sollten sich den Vorgaben und Absprachen gemäß (Kapitel 14) verhalten. Der OS/2-Kernel unterstützt eine Dateinummer nicht in spezieller Weise; bestimmte OS/2-System-Utilities (wie CMD.EXE und der *Presentation Manager*) folgen in ihrer Arbeitsweise diesen Konventionen.

Jeder Kindprozeß erbt alle Dateinummern des Vaterprozesses, solange dieser eine Nummer nicht explizit als *nicht vererbbar* kennzeichnet. Die Verwendung von STDIN, STDOUT und STDERR in einer vererbten Umgebung wurde bereits erläutert; was geschieht mit den anderen Dateinummern?

Obwohl ein Kindprozeß grundsätzlich alle Dateinummern des Vaterprozesses erbt, werden für gewöhnlich nur die Angaben für STDIN, STDOUT und STDERR gemacht. Die übrigen Nummern werden nicht zur Verarbeitung herangezogen. Lediglich die Nummernwerte 0 (STDIN), 1 (STDOUT) und 2 (STDERR) haben ihre besondere Bedeutung. Alle anderen Dateinummern sind lediglich Werte, die OS/2 für nachfolgende Ein-/Ausgabeaufrufe liefert. OS/2 garantiert keine besondere Reihenfolge in der Nummernvergabe; Anwendungen sollten daher niemals auf andere als die STD-Nummern zugreifen.

Hat beispielsweise ein Vaterprozeß eine Datei über die Nummer *N* geöffnet und ein Kindprozeß erbt diese Nummer, geschieht im allgemeinen nichts. Der Kindprozeß erhält den Wert *N* nicht als Resultat eines *DosOpen*-Aufrufs, da die Nummer bereits vergeben ist. Der Kindprozeß wird mit der Nummer *N* keine Operationen ausführen, da von ihm keine solche Nummer geöffnet wurde und er von ihrer Existenz keine Kenntnis hat.

Durch die Vererbung dieser überflüssigen Nummern treten zwei Nebeneffekte auf: Die durch die Nummer geöffnete Einheit kann nicht geschlossen werden, bis sowohl der Vaterprozeß als auch der Kindprozeß ihre Nummern freigeben. Da der Kindprozeß von der geöffneten Einheit nichts weiß, wird sie von ihm auch nicht geschlossen. Das Schließen einer Einheit durch den Vaterprozeß tritt daher solange nicht auf, bis sowohl der Kindprozeß als auch dessen Ablegerprozesse, die wiederum die Nummer erben, beendet worden sind. Falls eine andere Anwendung auf diese Datei oder Einheit zugreifen will, ist diese nicht verfügbar, da der Kindprozeß sie ohne Kenntnis der Anwendung geöffnet hält.

Der andere Nebeneffekt besteht darin, daß die Weitergabe von Nummern Speicherkapazität des Kindprozesses beansprucht. Obwohl das standardmäßige Maximum von 20 geöffneten Dateien einfach erhöht werden kann, kann bereits das Öffnen von nur zehn Dateien durch einen Kindprozeß die Erhöhung der Anzahl der möglichen geöffneten Dateien notwendig machen. Wenn der Prozeß einem Kindprozeß zwölf Dateien vererbt, reichen die verfügbaren Dateinummern nicht mehr aus, wenn nur zehn weitere Dateien geöffnet werden sollen. Das Schreiben von Programmen, die ihre Dateinummernbeschränkungen verändern, sollte vermieden werden, da diese Nummernweitergabe einen Overhead darstellen und das Problem der offengehaltenen und nicht mehr benötigten Dateien dadurch aufgebläht wird. Statt dessen sollten Vaterprozesse die Anzahl der unbenutzten Dateinummern, die an einen Kindprozeß vererbt werden, minimieren.

Beim Öffnen einer Datei durch ein Programm sollte dies mit der *DosOpen*-Anforderung und gesetztem *nicht vererbbar*-Bit geschehen, sofern die Datei für ein Kindprogramm nicht von Interesse ist. Falls dieses Bit beim Öffnen nicht gesetzt wird, kann das später mit dem Aufruf von *DosSetFHandState* erfolgen. Diese Bits werden für die Dateinummer (nicht für die Datei) gesetzt. Falls nun eine Datei über zwei Nummern geöffnet

ist, kann eine der Dateinummern vererbt werden, während die andere mit dem Attribut *nicht vererbbar* belegt ist. Diese Vorsichtsmaßnahme sollte unter allen Umständen auch bei Dateien verwendet werden, die keinen Kindprozeß in Gang setzen. So können *Dynlink*-Bibliotheksroutinen ohne Wissen des Benutzers Kindprozesse aufrufen. Auch in *Dynlink*-Programmen sollte das *nicht vererbbar*-Bit beim Öffnen von Dateien gesetzt werden, da das Vaterprogramm seinerseits Kindprozesse erzeugen kann.

Auch die UNIX-Praxis des Schließens aller Dateinummern von 3 bis 20 bei der Programminitialisierung sollte nicht übernommen werden. Die *Dynlink*-Untersysteme werden zur Initialisierung eines neuen Vaterprozesses aufgerufen, bevor die Kontrolle an die Anwendung übergeben wird. Falls Untersysteme bei der Initialisierung geöffnete Dateien besitzen, kann eine allgemeine Schließanweisung zu Ablauffehlern führen. Alle Programme verwenden die *Dynlink*-Untersysteme (ob diese tatsächlich angesprochen werden oder nicht, ist dabei gleichgültig), da sowohl das OS/2-Schnittstellenpaket als auch der *Presentation Manager* solche Untersysteme sind. Das versehentliche Schließen der Dateinummern eines Untersystems kann zu bizarren und vordergründig nicht erklärbaren Problemen führen. Bei der Initialisierung des VIO-Untersystems wird eine Nummer als Dateinummer angegeben, die auf die Bildschirmeinheit umgeleitet wird. Die Annahme, daß das Schließen der Nummer für Programme sicher ist, die das VIO-Paket nicht aufrufen, trifft nicht zu. Falls ein Schreibvorgang an STDOUT ausgegeben wird und die Standardausgabe auf die Bildschirmeinheit zeigt, ruft OS/2 den VIO für die Anwendung auf, was zu unvorhersehbaren Ergebnissen führt.

Bislang wurde erläutert, wie ein Kindprozeß durch die Vererbung der STDIN- und STDOUT-Nummern des Vaterprozesses Eingaben aus dem Eingabestrom des Vaterprozesses herausfiltert und seine Ausgabe mit dem Ausgabestrom des Vaterprozesses vermischt. Was hindert den Vaterprozeß am Lesen der für den Kindprozeß bestimmten Eingabe und was hindert den Vaterpro-

zeß am Überschreiben der durch den Kindprozeß ausgegebenen Daten?

Beim Öffnen einer Datei belegt OS/2 jedesmal eine Nummer für den Prozeß und nimmt einen entsprechenden Eintrag in der Nummerntabelle des Prozesses vor. Dieser Eintrag zeigt auf die *Systemdateitabelle* (SFT für *system file table*) in OS/2. Die SFT enthält einen Suchzeiger auf die Datei - das ist die Stelle der Datei, die momentan gelesen oder in die geschrieben wird. Wenn eine Nummer vererbt oder dupliziert wird, so zeigt die neue Nummer auf denselben SFT-Eintrag wie die Originalnummer. Falls beispielsweise die STDIN-Nummer des Kindprozesses einen Suchzeiger mit der STDIN-Nummer des Vaterprozesses teilt, so wird für beide derselbe Suchzeiger verwendet. Auf das Additionsbeispiel übertragen, bedeutet dies, daß durch das Lesen von zwei Zeilen durch NUMADD (aus STDIN) der Suchzeiger der eigenen STDIN-Nummer (und damit auch der Suchzeiger des Vaterprozesses, sowie der von dessen Vaterprozessen usw.) vorgerückt wird. Auch bei der Ausgabe zu STDOUT durch den Kindprozeß wird der Suchzeiger entsprechend weitergerückt, damit nachfolgende Schreibvorgänge durch den Vaterprozeß an die Ausgabe des Kindprozesses angehängt werden und diese nicht überschreiben.

Dieser Mechanismus unterliegt zwei Einschränkungen. Zum ersten darf in Situationen wie im Beipiel von NUMARITH und NUMADD der Vaterprozeß keine Ein-/Ausgabe über die STD-Nummern ausführen, bis der Kindprozeß seine Ein- bzw. Ausgabe vollendet hat. Zum zweiten muß bei der Ein- und Ausgabe von Daten sorgfältig auf die Datenpufferung geachtet werden.

Die meisten Programme lesen Daten aus STDIN bis ein EOF-Zeichen (*end of file*) gefunden wird. Solche Programme können STDIN beliebig puffern. Ein Programm wie NUMARITH, bei dem Kindprozesse Teile der STDIN-Daten lesen, kann keine Pufferung vornehmen, da die Pufferung die durch den Kindprozeß zu lesenden Daten beeinflussen könnte. NUMARITH kann die Daten nicht wieder in den Puffer schreiben, indem

der STDIN-Suchzeiger in umgekehrter Richtung verschoben wird, da STDIN auf eine Einheit (zum Beispiel die Tastatur oder eine Pipe) zeigen könnte, mit der keine Suchvorgänge ausgeführt werden können. Da NUMADD nur zwei Eingabezeilen lesen soll, muß das Lesen aus STDIN zeichenorientiert erfolgen, damit nicht versehentlich zusätzliche Daten gelesen werden.

Auch die Pufferung in STDOUT muß mit Bedacht vorgenommen werden. Grundsätzlich ist die Pufferung in der Standardausgabeeinheit möglich; bevor aber ein Kindprozeß auf die Standardausgabe zugreift, sollte der Pufferinhalt gelöscht sein.

Es besteht auch die Möglichkeit, daß ein Vaterprozeß die Standardnummern STDIN, STDOUT und STDERR nicht vererben will. Der Vaterprozeß sollte die Nummern in diesem Fall sicherlich nicht als *nicht vererbbar* kennzeichnen, da die Dateieinheiten für den Kindprozeß nicht geöffnet sind. Der OS/2-Kernel besitzt keine Routine, die die Standardeinheiten überwacht. Falls der Kindprozeß *DosOpen* ausführt und die Dateinummer 1 nicht gesetzt ist (da der Vaterprozeß für die Nummer 1 das *nicht vererbbar*-Attribut gesetzt hat), könnte OS/2 die neue Datei unter Angabe der Nummer 1 öffnen. Das Resultat besteht darin, daß die Ausgabe des Kindprozesses, die für STDOUT bestimmt ist, in eine andere Datei geschrieben wird (der zufälligerweise gerade die Dateinummer 1 zugeordnet ist).

Falls ein Kindprozeß die STD-Nummern nicht erben soll, sollte der Vaterprozeß die *DosDupHandle*/Umbenennen-Methode verwenden, und die Nummer auf die Null-Einheit zeigen lassen. Dies wird durch Öffnen einer Dateinummer auf die Null-Einheit realisiert und der Verschiebung der Nummer zu 0, 1 oder 2 mit *DosDupHandle*. Damit wird garantiert, daß alle Standardnummern für den Kindprozeß ansprechbar sind.

In Kapitel 14 (Interaktive Programme) werden STDIN, STDOUT und die Vererbung von Nummern ausführlicher behandelt.

4.2 PIDs und Befehlsunterbäume

Die PID (*process identification*) ist ein einzigartiger Code, der jedem Prozeß bei der Initialisierung unter OS/2 zugewiesen wird. Der Wert hat für die Prozesse keine Bedeutung, er ist lediglich eine Kennzeichnung (ein Name) für einen Prozeß. Eine PID kann jeden Wert außer Null annehmen. Ein PID-Wert wird niemals zwei Prozessen zum gleichen Zeitpunkt zugeordnet; PID-Werte können zwar wiederverwendet werden, dies geschieht jedoch nicht sofort, nachdem ein Prozeß mit einer PID-Nummer beendet wurde. Selbst wenn ein Kindprozeß mit einer Ihnen bekannten PID-Nummer bereits beendet ist und Sie diesen Prozeß kurze Zeit später aufrufen wollen, so wird die PID noch nicht wieder einem weiteren Prozeß zugeordnet sein. Es müssen circa 65.000 Prozesse initiiert werden, bevor dieser eine PID-Wert wiederverwendet wird. Selbst bei einem sehr stark benutzen PC-System wird vor der Wiederverwendung einer PID mindestens ein Zeitraum von einem Tag vergehen.

Bis zu diesem Punkt wurde die Architektur beschrieben, die Kindprogrammen die Initiierung weiterer Kindprozesse erlaubt. Dabei muß der Vaterprozeß die Struktur des Kindprozesses nicht kennen und somit auch nicht wissen, ob der Kindprozeß weitere Kindprozesse aufruft. Der Vaterprozeß braucht dieses Wissen nicht und soll es auch nicht haben, da die Informationen den Vaterprozeß von der laufenden Anwendung abhängig macht und dies unter OS/2 vermieden werden soll. Wie kann der Vaterprozeß nun feststellen, ob ein Kindprozeß (und dessen Ablegerprozesse) beendet ist? Das System kann dem Vaterprozeß mitteilen, ob der direkt aufgerufene Kindprozeß beendet ist (was aber nicht ausreichend ist). Der Kindprozeß kann andere Prozesse verursacht haben und vor der Ausführung seiner Kindprozesse beendet worden sein. Ebenso kann der Vaterprozeß beispielsweise die Priorität der erzeugten Prozesse ändern oder Kindprozesse beenden wollen, da ein Fehler aufgetreten ist oder der Benutzer mit der Tastenkombination Ctrl-C den Programmablauf unterbrochen hat.

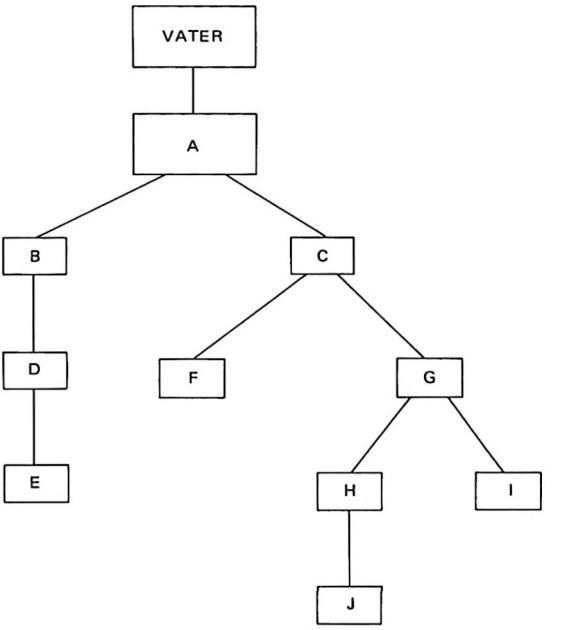


Bild 4.5 Befehlsunterbaum. A ist die Wurzel eines Befehlsunterbaumes des Vaterprozesses. B und C sind die Wurzeln von zwei Unterbäumen von A usw.

Diese Funktionen werden mit dem Konzept des *Befehlsunterbaumes* abgedeckt. Bei der Initiierung eines Kindprozesses wird dem Vaterprozeß die PID des Kindprozesses mitgeteilt. Die PID ist der Name des Kindprozesses und als Kennummer im Befehlsunterbaum ist sie der Name eines Astes von Ablegerprozessen, deren Wurzel der Kindprozeß ist. Wird die PID als Kennnummer eines Befehlsunterbaumes verwendet, so bezieht sie sich nicht auf den direkt initiierten Kindprozeß, sondern auch auf dessen Kindprozesse. Einem einzigen Ast eines solchen Baumes können Dutzende von Prozessen zugeordnet sein (Bilder 4.5 und 4.6).

OS/2-Funktionen (wie *DosCWait* und *DosKillProcess*) übernehmen entweder eine PID oder einen Befehlsunterbaumwert (abhängig von der angeforderten Unterfunktion). Wird die PID verwendet, so ist nur der genannte Prozeß angesprochen. Bei der Verwendung der Befehlsunterbaumform sind sowohl der genannte Prozeß als auch die Ableger betroffen. Dies gilt auch, wenn ein direkter Kindprozeß nicht mehr existiert oder wenn der Familienast der Prozesse Löcher aufgrund von Prozeßbeendigungen aufweist.

Bei der Verwendung der Befehlsunterbaumformen werden keine Beschränkungen gesetzt. Das bedeutet, daß ein Befehlsunterbaum X von OS/2 auch angesprochen werden kann, wenn der Kindprozeß X schon lange beendet ist. Konsequenterweise setzt OS/2 eine einfache Beschränkung bei der Verwendung von Befehlsunterbäumen, damit keine umfangreichen History-Einträge (Beschreibung der Veränderungen, die zur momentanen Systemsituation führen) gewartet werden müssen: Lediglich der direkte Vater des Prozesses X kann auf den Befehlsunterbaum X zugreifen. Mit anderen Worten: Der Großvaterprozeß von X kennt die PID des Prozesses X nicht und kann folglich auch keinen Einfluß auf Befehlsunterbäume von X nehmen. Dadurch wird eine Beschränkung im Umfang und der Dauer der Aufbewahrung von Befehlsunterbauminformationen in OS/2 gesetzt: Bei der Beendigung eines Prozesses werden die Informationen gelöscht, die sich auf dessen Unterbäume beziehen.

Das Befehlsunterbaumkonzept ist rekursiv. OS/2 löscht zwar die Informationen über die Befehlsunterbäume eines beendeten Prozesses, falls aber einer der Ablegerprozesse noch existiert, so verfügt dieser über Informationen bezüglich seiner Kindprozesse. Die "überlebenden" Ableger sind noch immer Teil des Befehlsunterbaumes, der zum Vaterprozeß des beendeten Prozesses gehört.

Hinweis: Dabei wird davon ausgegangen, daß der Vaterprozeß noch existiert. In jedem Fall sind alle Prozesse Teil einer geschachtelten Struktur von Befehlsunterbäumen, die zu den überlebenden Ahnenprozessen gehören.

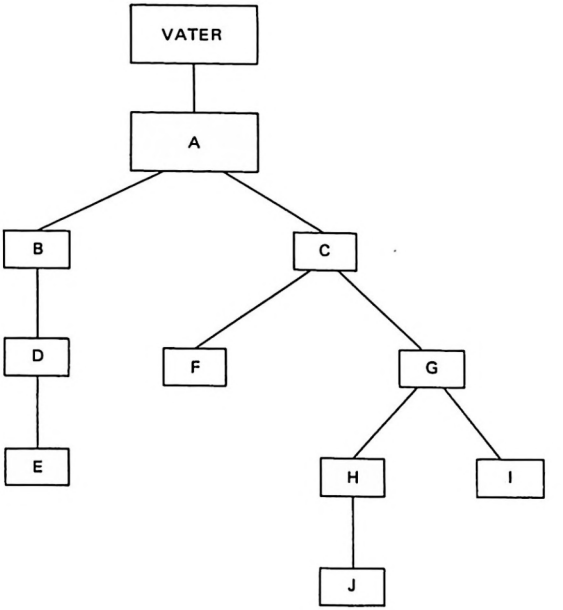


Bild 4.6 Befehlsunterbaum. Die grau unterlegt dargestellten Prozesse wurden beendet, die Unterbäume blieben jedoch erhalten. Unterbaum A kann von VATER verwendet werden, um die verbleibenden Unterprozesse anzusprechen. In gleicher Weise beeinflußt eine Operation über C für den Unterbaum G den Prozeß J.

4.3 DosExecPgm

Zur Ausführung eines Kindprozesses wird ein *DosExecPgm*-Aufruf ausgeführt. Die Form dieses Aufrufs finden Sie in Listing 4.3.

```
extern unsigned far pascal DOSEXECPGM (
        char far                        *OBJNAMEBUF,
        unsigned                        OBJNAMEBUFL,
        unsigned                        EXECTYPE,
        char far                        *ARGSTRING,
        char far                        *ENVSTRING,
        struct ResultCodes far  *CODEPID,
        char far                        *PGMNAME);
```

Listing 4.3 Aufruf von DosExecPgm

Durch die Verwendung der *objnamebuf*-Argumente wird ein Speicherbereich automatisch zur Verfügung gestellt, in dem OS/2 eine Zeichenkette ablegen kann, wenn der *DosExecPgm*-Aufruf sich nicht ausführen läßt. Unter MS-DOS Version 2.0 und höher war die *Exec*-Funktion recht einfach: Eine Datei wurde in den Speicher geladen. Dabei konnten nur wenige Fehler auftreten: entweder wurde die Datei nicht gefunden, es lag ein falsches Dateiformat vor oder der zur Vergügung stehende Speicherbereich war unzureichend. Ein einfacher Fehlercode genügte zur Diagnose eines Problems. Unter OS/2 ist dieses Problem durch die dynamische Verknüpfung vielfältiger. Bei früheren Prototypversionen des Betriebssystems OS/2 wurden die *objnamebuf*-Argumente nicht verwendet und die Softwareentwickler erhielten häufig mit einer Fehlermeldung den Fehlercode *Dynlink Library Load Failed*. In welcher Bibliothek liegt aber nun der Fehler? Die Anwendung bezog sieben Bibliotheken ein, von denen keine einen offensichtlichen Fehler aufzeigte. Da sich die Bibliotheken auf weitere Bibliotheken beziehen können, ist die Struktur schwer überschaubar: Es wurden Objektnamenargumente zur besseren Kennung hinzugefügt. Der Puffer ist der Bereich, in den OS/2 den Namen der fehlenden oder nicht funktionsfähigen Bibliothek schreibt und das Längenargument gibt OS/2 dabei die maximale Größe des Pufferbereichs an. Strings, die die Größe dieses Bereiches überschreiten, werden abgeschnitten.

Das Wort *Exectype* beschreibt die Formen des *DosExecPgm*-Aufrufs. Folgende Werte können hierfür verwendet werden:

- *0: Synchrone Ausführung des Kindprogrammes.* Der den *DosExecPgm*-Aufruf ausführende Thread wird nicht weiter bearbeitet, bevor der Kindprozeß nicht beendet ist. Die weitere Programmausführung geschieht erst dann, wenn der Kindprozeß beendet wird. Die Beendigung aller Programme eines Befehlsunterbaumes ist dabei nicht Bedingung. Zur Vereinfachung der Programmkonvertierung von MS-DOS- zu OS/2-Versionen wurde diese Form des *DosExecPgm*-Aufrufs implementiert. Sie sollten die Verwendung dieses Aufrufs vermeiden, da der ordnungsgemäße Programmablauf von der Benutzung der Tastenkombination Ctrl-C und Ctrl-Break sehr stark beeinträchtigt werden kann (siehe Kapitel 14, Interaktive Programme).

- *1: Asynchrone Programmausführung.* Der Kindprozeß wird ausgeführt, sobald der Scheduler die Möglichkeit dazu anbietet; der aufrufende Thread arbeitet sofort nach dem *DosExecPgm*-Aufruf weiter. Es kann nicht als sicher angenommen werden, daß dem Kindprozeß bereits Rechenzeit zugeteilt wurde, nachdem der Vater-Thread den Aufruf von *DosExecPgm* ausgeführt hat; andererseits kann aber auch nicht davon ausgegangen werden, daß der Kindprozeß noch keine Rechenzeit zu diesem Zeitpunkt erhalten hat. Mit dieser Aufrufform wird OS/2 angewiesen, sich nicht um Beendigungscodes des Kindprozesses für weitere *DosCWait*-Aufrufe zu kümmern. Die Aufrufe werden benötigt, wenn der Vaterprozeß sich um das Ergebnis des Kindprozesses, auf die Beendigung oder allgemein das weitere Verhalten nicht bezieht oder benötigt. Beim Aufruf von Dienstleistungsprogrammen wird diese Programmaufrufform selten verwendet. Als System-Utility (zum Beispiel zur periodischen Speicherung veränderter Daten) kann die Aufrufform eingesetzt werden. Beachten Sie, daß ein auf diese Weise initialisierter Prozeß im Gegensatz zum *abgehängten Prozeß* (der im folgenden beschrieben wird) als Kindprozeß des aufrufenden Vaterprozesses definiert ist.

Der Vaterprozeß kann einen *DosCWait*-Auruf ausführen, um festzustellen, ob der Kindunterbaum noch bearbeitet wird oder bereits beendet ist. Es wird bei der Beendigung des Kindprozesses allerdings kein Rückgabecode übergeben.

- *2:* Dieser Aufruf entspricht dem Aufruf der Form 1 insofern, als daß von einem Programm ein Kindprozeß asynchron ausgeführt wird. OS/2 hält dabei aber den Rückgabecode als Antwort des Kindprozesses für eine zukünftige Nachfrage durch einen *DosCWait*-Aufruf bereit. Auf diese Weise kann ein Programm herausfinden, ob der abgesetzte Programmaufruf (woraus ein Programmstart resultiert) zu einem ordentlichen Ablauf geführt hat; damit wird der Status eines Kindprozesses bestimmt. Der Vaterprozeß sollte einen *DosCWait*-Aufruf baldmöglichst ausführen, da OS/2 Version 1.0 vorsichtshalber für einen Kindprozeß, der mit *DosExecPgm* gestartet wurde, über 2600 Bytes Daten über einen beendeten Prozeß bereithält, um auf einen möglichen *DosCWait*-Aufruf Antwort geben zu können. Die Zwischenlagerung solcher Daten für eine gewisse Zeit sollte kein Problem darstellen, eine Speicherung über mehrere Stunden hinweg sollte hingegen vermieden werden.

  Da OS/2 alle möglichen Eventualitäten einzubeziehen versucht, kann vom Benutzer der *DosCWait*-Aufruf sowohl vor oder nach der Beendigung des Kindprozesses ausgegeben werden. Obwohl ein Vaterprozeß die relative Zuweisung der Rechenzeit zwischen Kind- und Vaterprozeß durch das Setzen der Priorität festlegen kann, kann nicht verläßlich festgelegt werden, welcher Prozeß zu welchem Zeitpunkt bearbeitet wird. Aus diesem Grunde sollten Programme so entwickelt werden, daß die Beendigung von Kindprozessen von der Abarbeitung und Beendigung der Vaterprozesse unabhängig ist. Alternativ kann eine Form des IPC zur Ausführungssynchronisation von Vater- und Kindprozeß verwendet werden (siehe 4.4 *DosCWait*, dort finden Sie weitere Details).

- *3: Diese Form wird vom Systemdebugger CodeView verwendet.* Das Betriebssystem unterbindet das Einklinken eines Prozesses in einen beliebigen anderen Prozeß, damit dieser die Ausführung eines zweiten Prozesses beeinflussen kann. Solche Eingriffsmöglichkeiten eines Prozesses auf einen anderen könnten zu schwerwiegenden und nicht mehr kontrollierbaren Nebeneffekten führen - es widerspricht der Vereinzelung der Prozesse, auf der das Betriebssystem aufgebaut ist. Hinzu kommt, daß damit von OS/2 keine Kontrolle über ablaufende Prozesse ausgeübt werden könnte. Die für Programme zur Verfügung gestellte Umgebung für den reibungslosen Ablauf wäre nicht vorhanden. Unter OS/2 wird diese Sicherheit für einen kontrollierten Programmablauf mit der *DosPtrace*-Funktion gewährleistet, mit der der Kindprozeß kontrolliert wird. Dies geschieht, indem Daten aus Speicherstellen gelesen und in Speicherstellen geschrieben werden. Es können nur dann Prozesse kontrolliert werden, wenn diese in einem Spezialmodus mit dem Wert 3 ausgeführt worden sind.

- *4: Mit diesem Befehlsaufruf wird der Kindprozeß asynchron ausgeführt; der auszuführende Prozeß, der mit diesem Aufruf gestartet wird, hat keine Verbindung zu dem Prozeß, der den Start auslöst.* Ein auf diese Weise "abgehängter" Prozeß erbt nicht die Bildschirmgruppe (Festlegung der Ausgabeeinheit) des Vaterprozesses, sondern wird in einer sogenannten *ungültigen Sonderbildschirmgruppe* ausgeführt. Jeder Versuch einer Bildschirmausgabe, einer Tastatur- oder Mauseingabe auf Basis der angegebenen Geräteeinheit führt zur Rückgabe eines Fehlercodes. Das System sieht den abgehängten Prozeß nicht als Kind des Vaterprozesses an. Der neue Prozeß besitzt zu dem aufrufenden Prozeß keine Verknüpfung mehr - mit anderen Worten: er ist vaterlos. Diese Form des *DosExecPgm*-Aufrufs wird zur Initiierung verborgener Programme verwendet, die ohne direkte Interaktion mit dem Benutzer ausgeführt werden sollen.

Das Argument *EnvString* zeigt auf eine Liste von ASCII-Zeichenketten, die Werte zur Umgebungskonfigurierung enthalten. MS-DOS unterstützt lediglich einen einzelnen Umgebungsblock für das gesamte System. Bei der Verwendung nur eines Umgebungsblockes wäre die nicht akzeptable Möglichkeit der Querverbindung zwischen Anwendungen bereitgestellt. Aus diesem Grund unterstützt OS/2 einen eigenen Umgebungsblock für jeden Prozeß. Ein Prozeß erbt üblicherweise die Umgebung des Vaterprozesses. In diesem Fall sollte das Argument *EnvPointer* NULL sein, wodurch OS/2 zur Übergabe einer genauen Kopie der Umgebungsstrings des Vaterprozesses veranlaßt wird (siehe Listing 4.4).

```
PATH=C:\DOS;C:\EDITOR;C:\TOOL;C:\XTOOL;C:\BIN;C:\UBNET
INCLUDE=\include
TERM=h19
INIT=c:\tmp
HOME=c:\tmp
USER=c:\tmp
TEMP=c:\tmp
TERM=ibmpc
LIB=c:\lib
PROMPT=($p)
```

Listing 4.4 Typische Umgebungsstrings

Die Zeichenkettenangaben für die Festlegung der Programmumgebung werden zur Anpassung einer Ausführungsumgebung eingesetzt. Falls ein Benutzer zwei Bildschirmgruppen anlegt, läuft jede unter einer Kopie von CMD.EXE ab. Jedes CMD-.EXE ist ein direkter Kindprozeß des *Presentation Manager*, der Bildschirmgruppen verwaltet und anlegt. Die Prozesse sind gleichzeitig die Ahnen aller Prozesse, die in dieser Bildschirmgruppe ausgeführt werden. Falls der Benutzer Utility-Programme ausführt, die den Umgebungsstring *TEMP=<Verzeichnisname>* zur Spezifikation eines Verzeichnisses für Temporärdateien verwendet, können verschiedene Werte für den Umgebungsstring TEMP für jede Kopie von CMD.EXE gewollt sein. Das Ergebnis besteht darin, daß die Utilities das korrekte Verzeich-

nis als Temporärspeicher verwenden (abhängig von der Bildschirmgruppe, in der sie ausgeführt werden). Sie haben von ihren CMD.EXE-Ahnen den entsprechenden Umgebungsstring geerbt (siehe Kapitel 10 Umgebungsstrings).

Wegen der globalen Natur der Umgebungsstrings sollten Vaterprozesse bei der Editierung einer Liste der Umgebungsstrings lediglich solche Strings ändern oder löschen, die von Ihnen verwendet werden; nicht bekannte Zeichenketten sollten unverändert bleiben.

4.4 DosCWait

Bei der Ausführung eines Kindprozesses sollte dem aufrufenden Prozeß bekannt werden, ob und mit welchem Erfolgsstatus ein Kindprozeß beendet wurde. Die OS/2-Funktion *DosCWait* liefert diese Informationen. Vor der detaillierten Erklärung von *DosCWait* sind zwei Feststellungen zu machen. Obwohl *DosExecPgm* lediglich einen einzigen Prozeß startet, ist es möglich, daß dieser Kindprozeß wiederum eigene Kindprozesse initiiert usw. Ein Programm kann nicht als sicher annehmen, daß keine weiteren "Enkelkindprozesse" initiiert werden; statt dessen sollte das Programm die Befehlsunterbaumform von *DosCWait* verwenden. Der Rückgabecode des direkten Kindprozesses (das ist der Stamm des Befehlsunterbaumes) genügt, da der direkte Kindprozeß beim Aufruf anderer Prozesse diese überwachen und deren Erfolgsstatus mit *DosCWait* feststellen muß. Mit anderen Worten: Falls ein Kindprozeß einem Prozeß eine Aufgabe überträgt und dieser fehlerhaft beendet wird, muß auch der Kindprozeß mit einem Fehlercode beendet werden.

Zum zweiten ist zu beachten, daß es nicht immer nur einen Kindprozeß geben muß; es können mehrere Befehlsunterbäume gleichzeitig existieren. Der Vaterprozeß kann mit Kindprozessen über verschiedene anonyme Pipes verbunden sein oder Kindprozesse können unabhängig voneinander ablaufen. Die Ausführung eigener *DosCWait*-Aufrufe für jeden Prozeß oder Unterbaum ist nicht nötig.

Die Form des *DosCWait*-Aufrufs finden Sie in Listing 4.5.

```
extern unsigned far pascal DOSCWAIT (
        unsigned                    ACTIONCODE,
        unsigned                    WAITOPTION,
        struct ResultCodes far      *RESULTWORD,
        unsigned far                *PIDRETURN,
        unsigned                    PID);
```

Listing 4.5 Die DosCWait-Funktion

Drei der Argumente beziehen sich auf den Gültigkeitsbereich eines Befehls: *ActionCode*, *WaitOption* und PID. Am einfachsten wird deren Bedeutung klar, wenn die möglichen Werte in einer Tabelle zusammengefaßt werden.

DosCWait-Format: Befehlsunterbaum

Diese Form des *DosCWait*-Aufrufs verarbeitet die gesamte Befehlsunterbaumstruktur, die auch aus nur einem Kindprozeß bestehen darf. Wir empfehlen die Verwendung dieser Aufrufform, da sie auch nach einer Veränderung eines Kindprozesses (Aufruf von mehr oder weniger Kindprozessen) noch korrekt funktioniert.

ActionCode	WaitOption	PID	Handlung
1	0	n	Warten, bis der Befehlsunterbaum vollständig ausgeführt ist und Rückgabecode des direkten Kindprozesses liefern.
1	1	n	Rückgabe des Beendigungscodes des direkten Kindprozesses, nachdem der Befehlsunterbaum vervollständigt wurde. Rückgabe des Fehlercodes ERROR_WAIT-_NO_CHILDREN.

DosCWait-Format: einzelne Prozesse

Diese Form des *DosCWait*-Aufrufs wird zur Überwachung einzelner Kindprozesse verwendet. Die Prozesse müssen direkte Kindprozesse eines Prozesses sein, deren Nachfolger und nicht miteinander in Beziehung stehende Prozesse können mit dieser Funktion nicht überwacht werden. Verwenden sollten Sie diese Form nur dann, wenn der Kindprozeß Teil derselben Anwendung oder desselben Softwarepakets wie der aufrufende Prozeß ist; der Programmierer muß sicher sein, daß Prozesse, die vom Kindprozeß initiiert wurden, nach der Beendigung des direkten Kindprozesses nicht mehr abgearbeitet werden.

ActionCode	WaitOption	PID	Handlung
0	0	0	DosCWait kehrt zurück, sobald der direkte Kindprozeß beendet ist. Falls ein Kindprozeß zum Aufrufzeitpunkt bereits beendet ist, kehrt DosCWait direkt zurück.
0	0	N	DosCWait kehrt zurück, sobald der direkte Kindprozeß N beendet ist. Falls der Kindprozeß zum Aufrufzeitpunkt bereits beendet ist, kehrt DosCWait direkt zurück.
0	1	0	DosCWait prüft auf einen beendeten, direkten Kindprozeß. Falls ein beendeter Kindprozeß gefunden wird, wird dessen Status geliefert. Ein Fehlercode wird zurückgegeben, falls kein

			beendeter Kindprozeß gefunden wird.
0	1	N	DosCWait prüft den Status eines direkten Kindprozesses. Falls der Kindprozeß bereits beendet ist, wird dessen Status geliefert. Ein Fehlercode wird zurückgegeben, falls der Kindprozeß sich noch in der Ausführung befindet.

Hinweis: Es wird nicht als Fehler angesehen, den Beendigungscode eines Kindprozesses mit DosCWait zu bestimmen, wenn der Kindprozeß noch arbeitende Nachfolgeprozesse besitzt. Das bedeutet aber im allgemeinen, daß die Aufgabe des Kindprozesses noch nicht beendet ist.

DosCWait-Format: Nicht empfehlenswert

ActionCode	WaitOption	PID	Handlung
1	0	0	DosCWait wartet, bis ein direkter Kindprozeß und dessen Abkömmlinge beendet sind. Es wird der Rückgabecode des direkten Kindprozesses geliefert. Es wird nicht auf die Beendigung des ersten Befehlsunterbaumes gewartet, sondern ein Befehlsunterbaum aufgrund des ersten beendeten direkten Kindprozesses gewählt und dessen Beendigung erwartet (auch wenn andere Befehlsunterbäu-

			me zwischenzeitlich beendet wurden).
1	1	0	Diese Form liefert den Fehlercode ERROR_CHILD_NOT_COMPLETE, wenn noch mindestens ein Prozeß in einem beliebigen Befehlsunterbaum aktiv ist. Falls die Ausführung aller Unterbäume beendet ist, liefert der Aufruf den Rückgabecode des direkten Kindprozesses. Falls kein direkter Kindprozeß einen Rückgabecode besitzt, wird der Code ERROR_WAIT_NO_CHILDREN geliefert.

4.5 Steuerung von Kind-Tasks und Befehlsunterbäumen

Der Vaterprozeß hat nur eine begrenzte Kontrolle über seine Kindprozesse, da Nebeneffekte und Querverbindungen zwischen Prozessen im System unterbunden wurden. Der Vaterprozeß kann Befehlsunterbäume auf zwei Arten beeinflussen: er kann deren CPU-Priorität verändern oder sie beenden. Auch hier wird die Befehlsunterbaumform beider Befehle empfohlen, da diese gegenüber den Ausführungsdetails der Kindprozesse nicht sensitiv ist.

4.5.1 DosKillProcess

Nachdem ein Vaterprozeß einen Kindprozeß oder einen Befehlsunterbaum gestartet hat, kann dieser vor der normalen Beendigung abgebrochen werden. Dies erfolgt häufig aufgrund einer Benutzereingabe (beispielsweise der Tastenkombination Ctrl-Break). In Kapitel 14 (Interaktive Programme) sind Möglichkeiten zur Bearbeitung der Tastenkombinationen Ctrl-Break und Ctrl-C aufgeführt.

DosKillProcess markiert jeden Prozeß in einem Befehlsunterbaum (oder den direkten Kindprozeß, falls diese Aufrufform verwendet wird) zur Beendigung. Ein zur Beendigung markierter Prozeß wird beendet, nachdem alle seine Threads den Systemkern verlassen haben (das heißt, alle Threads sind von Systemaufrufen zurückgekehrt). Das System bricht Aufrufe ab, die die Prozeßbeendigung für mehr als eine Sekunde verzögern würden (beispielsweise das Lesen eines Zeichens von der Tastatur). Ein Prozeß kann SIGKILL zur verzögerten Beendigung (über eine beliebige Zeitspanne) verwenden. Die Realisierung einer Verzögerung über *SetSignalHandler*/SIGKILL ist nicht empfehlenswert und wird mehr als Fehler denn als Leistungsmerkmal angesehen.

4.5.2 DosSetPrty

Ein Kindprozeß erbt die Prozeßpriorität des Vaterprozesses bei einem *DosExecPgm*-Aufruf. Auch nach dem Aufruf kann der Vaterprozeß die Priorität des Befehlsunterbaumes bzw. des direkten Kindprozesses beeinflussen. Hier wird wieder die Befehlsunterbaumaufrufform mit folgender Maßgabe empfohlen: Wenn die Aufgabe des Kindprozesses die Priorität *N* besitzt, sollten alle dessen Kindprozesse ebenfalls mit der Priorität *N* ablaufen.

Kapitel 5

Threads und Scheduler-Prioritäten

5.1 Threads

Ein Programm ist nichts anderes als eine Folge von Befehlen, die im Hauptspeicher (RAM) abgelegt sind. Die CPU liest alle Befehle sequentiell und führt sie Schritt für Schritt aus. Die Bearbeitung einer Befehlsfolge durch die CPU wird *Thread of Execution* genannt. Alle Versionen von MS-DOS führen Programme aus, so daß sie notwendigerweise einen *Thread of Execution* unterstützen. OS/2 unterscheidet sich dadurch, daß mehrere *Threads of Execution* in einem einzigen Prozeß unterstützt werden. In anderen Worten: Mehrere Teile eines Programmes können zum gleichen Zeitpunkt ausgeführt werden.

Es ist offensichtlich, daß ein Multitasking-System mehrere Threads während des Abarbeitens eines Programms im Hauptspeicher unterstützen muß. Jeder Prozeß muß also mindestens einen Thread besitzen. Wenn gleichzeitig zehn Prozesse aktiv sind, müssen auch (mindestens) zehn Threads vorhanden sein. Eine solche Koexistenz mehrerer Threads in einem System ist für den Programmierer nicht direkt nachvollziehbar, da jedes Programm mit nur einem Thread abgearbeitet wird. Die Arbeitsweise von OS/2 unterscheidet sich von der Einzelprogrammbearbeitung in einer Singletasking-Umgebung (Anweisung für Anweisung), da hier ein einzelnes Programm mit mehreren Threads ausgeführt werden kann.

Da Threads Elemente eines Prozesses sind, und ein Prozeß quasi ein "Eigentümer" von Ressourcen ist, teilen alle Threads, die zu

einem Prozeß gehören, die Ressourcen dieses Prozesses. Falls ein Thread eine Datei mit der Dateinummer X öffnet, können alle Threads dieses Prozesses *DosRead*- und *DosWrite*-Aufrufe mit dieser Dateinummer ausführen. Belegt ein Thread ein Speichersegment, so können alle in diesen Prozeß einbezogenen Threads auf das Speichersegment zugreifen. Threads können jederzeit beendet werden, ohne daß dies von außen ersichtlich ist. Lediglich ein Thread ist zum ordentlichen Programmablauf notwendig, wenn auch dieser letzte Thread beendet wird, ist das Programm, das die Threads ins Leben rief, beendet.

Obwohl der Prozeß der Teil eines Programms ist, dem die Ressourcen zugeordnet sind, besitzt jeder Thread eigene, private Informationen. So verfügt jeder Thread über eine Kopie der CPU-Registerinhalte. Dies ist offensichtlich erforderlich, da jeder Thread in der Lage sein muß, unterschiedliche Befehlssequenzen abzuarbeiten. Threads müssen auch eine eigene Kopie der Fließkommaregister besitzen.

OS/2 legt den ersten Thread eines Prozesses bei Programmausführungsbeginn fest. Weitere Threads werden mit einem *DosCreateThread*-Aufruf initiiert. Jeder Thread kann weitere Threads aufrufen. Die Threads eines Prozesses werden als untereinander gleichwertig angesehen, zwischen ihnen besteht keine Vater-Kind-Beziehung, wie das bei Prozessen der Fall ist. Thread 1 besitzt spezielle Merkmale, die im folgenden erläutert werden.

5.1.1 Thread-Stapel

Jeder Thread besitzt einen eigenen Stapelbereich, auf den die SS- und SP-Werte des Threads zeigen. OS/2 belegt Stapelbereiche gemäß den Spezifikationen in der .EXE-Datei. Bei der Initiierung weiterer Threads über einen *DosCreateThread*-Aufruf muß das aufrufende Element einen Stapelbereich für den neuen Thread spezifizieren. Da der Speicher jedes Threads dem übergeordneten Prozeß zugewiesen ist, kann ein beliebiger Thread des Prozesses diesen Speicherbereich modifizieren. Der Programmierer hat dafür zu sorgen, daß dies nicht geschieht. Die

Größe des Segmentes, in dem der Stapel vorliegt, muß explizit angegeben werden, der Umfang des Thread-Stapels nicht. Der Programmierer kann den Stapel des Threads in dessen eigenem Segment oder in einem Datensegment (einschließlich anderer Thread-Stapel) verwenden. In jedem Fall muß genügend Stapelbereich für den Thread bereitgestellt werden. Zudem müssen zu jedem Stapelbereich eines Threads mindestens zwei Kbyte freier Speicherbereich hinzugefügt werden. Dieser Extraspeicher wird für die Verwendung der dynamischen Verknüpfungsroutinen benötigt, von denen einige einen umfangreichen Stapel benötigen. Alle Threads müssen diesen Speicherbereich reservieren (auch wenn sie die *Dynlink*-Routinen nicht aufrufen). Durch einen Hardwarefehler in vielen 80286-Prozessoren müssen Stapelsegmente von Anfang an mit ihrer vollen Größe belegt werden. Es dürfen nicht mehr Daten in ein Stapelsegment geschrieben werden, als verfügbare, vorbelegte Kapazität vorgesehen ist (auch nicht unter der Annahme, daß OS/2 den Speicherbereich ausdehnen wird). Dies führt zu einem Stapelsegmentfehler und der Prozeß wird beendet.

5.1.2 Verwendung von Threads

Es gibt viele Einsatzmöglichkeiten für Threads. In diesem Abschnitt finden Sie vier Beispiele, die Ihnen eine Vorstellung geben sollen.

5.1.2.1 Vorder- und Hintergrundarbeit

Threads stellen eine Art des Multitasking innerhalb eines Programms dar. Es bietet sich an, Threads zur simultanen Abarbeitung von Vorder- und Hintergrundaufgaben in einem Programm einzusetzen. Die Ausdrücke *Vordergrund* und *Hintergrund* sind hier im Sinne der Interaktion mit dem Benutzer verwendet und beziehen sich nicht auf Vorder- und Hintergrundbildschirmgruppen. So kann ein Tabellenkalkulationsprogramm beispielsweise einen Thread zur Anzeige von Menüs und der Entgegennahme der Benutzereingabe verwenden, während ein zweiter Thread die vom Benutzer eingegebenen Befehle aus-

führt, die Kalkulationstabelle aktualisiert und andere Aufgaben erledigt.

Diese Nutzung erhöht im allgemeinen die Geschwindigkeit, mit der ein Programm auf Benutzereingaben reagiert. Falls der Benutzer beipielsweise den Inhalt einer Zelle verändert und eine Neuberechnung veranlaßt, kann der *Ausführung*-Thread die Neuberechnung durchführen, während der *Befehl*-Thread dem Benutzer die Verschiebung des Cursors, die Auswahl neuer Menüs und andere Tätigkeiten erlaubt. Die Tabellenkalkulation sollte RAM-Semaphoren zum Schutz ihrer Strukturen verwenden, so daß ein Thread nicht auf eine Struktur zugreifen kann, die derzeit durch einen anderen Thread manipuliert wird. Der Benutzer braucht auf die Thread-Folge keine Rücksicht zu nehmen. Der letzte eingegebene Befehl ist im allgemeinen vollständig abgearbeitet, bevor der Benutzer einen neuen Befehl eingegeben oder gewählt hat. Falls dies nicht der Fall sein sollte, wird die Ausführung des neuen Befehls bis zur Komplettierung des ersten Befehls verzögert. Dies kann vorkommen, wenn der Benutzer einer Tabellenkalkulation eine Zeile direkt nach dem Speichern der Kalkulationstabelle auf Diskette löscht. Der Durchsatz in dieser *Worst-Case*-Situation (dem schlimmsten, denkbaren Fall) entspricht der Geschwindigkeit, die mit einem Ein-Thread-Konzept erreicht wird.

5.1.2.2 Asynchrone Verarbeitung

Eine andere Einsatzmöglichkeit von Threads besteht in der Bereitstellung asynchroner Elemente im Programmkonzept. So kann beispielsweise zum Schutz gegen den Stromausfall ein Textverarbeitungsprogramm so entwickelt werden, daß der bearbeitete Inhalt (Text) im RAM-Puffer einmal pro Minute auf Diskette oder Platte übertragen wird. Threads machen die Zeitspanne, die zur Überprüfung der Systemuhr verwendet wird, unnötig und verhindern, daß eine Abspeicherung ausfällt, weil das Programm in einer Warteschleife (etwa dem Lesen von der Tastatur) hängengeblieben ist. Es kann ein Thread angelegt werden, dessen einzige Aufgabe die periodische Sicherung des Textes ist. Der Thread kann die Funktion *DosSleep* für 60 Se-

kunden aufrufen, anschließend den Pufferinhalt auf Diskette oder Platte schreiben, einen weiteren *DosSleep*-Aufruf für 60 Sekunden ausführen usw.

Das asynchrone Ereignis muß nicht zeitlich veranlaßt sein. So kann in einem Kommunikationsprogramm bei der Verwendung eines asynchronen seriellen Ports ein Thread auf ein Timeout des Netzwerkprotokolls warten. Der Haupt-Thread kann die Interaktion mit dem Benutzer in dieser Zeit fortführen.

Auch Programme, die anderen Programmen Dienstleistungen im Rahmen der IPC anbieten, können Threads zur gleichzeitigen Ausführung mehrerer Anfragen verwenden. So kann beispielsweise ein Thread die Arbeitswarteschlange verwalten, während andere Threads die eigentlichen Aufgaben ausführen.

5.1.2.3 Beschleunigte Ausführung

Durch die Verwendung mehrerer Threads kann die Ausführungsgeschwindigkeit eines Prozesses durch die Überlappung von Ein-/Ausgabe und Berechnungen verkürzt werden. Ein Prozeß mit nur einem Thread kann Berechnungen ausführen und OS/2-Aufrufe für das Lesen oder Schreiben von/auf Disketten ausführen - aber nicht beides gleichzeitig. Ein Prozeß mit mehreren Threads hingegen kann beides gleichzeitig, also beispielsweise einen Datenstapel verarbeiten, während der nächste Stapel aus einer Einheit gelesen wird.

In Zukunft werden sicherlich PCs mit mehreren 80386-Prozessoren verfügbar werden (Multiprozessorsysteme). Eine Anwendung, die mehrere Threads verwendet, kann durch die gleichzeitige Verwendung mehrerer Prozessoren noch schneller werden.

5.1.2.4 Programmstruktur

Threads können auch zur Strukturierung von Programmen bzw. zur Vereinfachung einer Programmstruktur verwendet werden. So kann beispielsweise in einem Sicherheits-/Alarmsystem ein Thread für jede Aktivität eingesetzt werden. Ein Thread kann

den Status der Sicherheitsschalter abfragen, während ein zweiter Thread die Lichter kontrolliert, ein dritter für die Telefonverbindung verantwortlich ist und ein weiterer Thread die Benutzerschnittstelle verwaltet.

Diese Struktur vereinfacht das Softwarekonzept. Der Programmierer braucht sich nicht um die unbemerkte Umschaltung eines Schalters zu kümmern, während die CPU anderweitig beschäftigt ist (beispielsweise den zweiten Tastenanschlag eines 2-Tasten-Befehls erwartet). Auch die Verwendung mehrerer Konsolen wird einfacher, da jede Konsole von einem eigenen Thread unterstützt werden kann, der lokale Variablen (im Stapel) unterhält. Mehrere Konsolen können also ohne Probleme gleichzeitig verwendet werden.

Natürlich kann ein derartiges Programm auch mit nur einem Thread realisiert werden; in diesem Fall sind aber mehrere Ereignisflaggen und Abfrageschleifen nötig. Eine andere Möglichkeit ist die Verwendung von Unterroutinen. Am besten und einfachsten ist aber die Verwendung mehrerer Threads.

5.1.3 Zu berücksichtigende Grenzen

Ein Vorteil bei der Verwendung von Threads besteht darin, daß die Daten, Dateien und Ressourcen eines Prozesses von allen zugehörigen Threads geteilt werden. Dies stellt allerdings auch gleichzeitig einen Nachteil dar, weil diese Elemente vor der gleichzeitigen Aktualisierung durch mehrere Threads geschützt werden müssen. Heutige OS/2-Rechner verfügen im allgemeinen nur über einen einzigen Prozessor. Die scheinbar wahlfreie Zuteilung der Rechenzeit durch den Scheduler schafft die Illusion der gleichzeitigen Ausführung der Threads. Selbstverständlich ist die Zuteilung der Rechenzeit nicht wahlfrei, da der Scheduler auf die Priorität der einzelnen Prozesse achtet. Zur Realisierung eines guten Programmierstils sollte die Verteilung der Rechenzeit aber vollständig als Zufallsfaktor angesehen werden. Bei der Ausführung eines Programms nehmen externe Ereignisse auf den Scheduler Einfluß. So kann beispielsweise ein Task mit höherer Priorität die CPU in Anspruch nehmen.

Oder der Schreib-/Lesekopf des Diskettenlaufwerks ist in einer anderen Position und ein Lesevorgang von der Diskette benötigt dadurch mehr Zeit als üblich. Es kann nicht einmal mit Sicherheit angenommen werden, daß der Thread mit der höchsten Priorität ausgeführt wird, da bei einem Mehr-CPU-System *N* Threads mit der höchsten Priorität gleichzeitig ausgeführt werden.

Die einzig sichere Annahme kann sein, daß alle Threads gleichermaßen und gleichzeitig ausgeführt werden und daß - in Abwesenheit einer expliziten Sperre oder Semaphoren-Aufrufe - jeder Thread immer das tut, was dem *Worst Case* bei der Aktualisierung statischer Daten und Strukturen gleichkommt. Die Entwicklung eines Programms unter der Annahme, daß bestimmte Threads zu bestimmten Zeitpunkten ausgeführt werden, ist unbedingt zu vermeiden. Das Austesten eines solchen Programmes ist unmöglich, zudem gemäß den Gesetzen von Murphy die Probleme während des Testens nur selten auftreten, dafür aber umso häufiger bei der täglichen Arbeit mit dem Programm.

5.1.3.1 Lokale Variablen

Um die Gefahr einer Kollision von Threads bei der Bearbeitung statischer Daten zu minimieren, sollten in einem Programm nur die Mindestmenge statischer Daten verwendet werden. Wie jeder Thread einen eigenen Stapel besitzt, so verfügt er auch über einen eigenen *Stapel-Frame*, in dem lokale Variablen abgelegt werden.

Wenn beispielsweise ein Thread eine Datei öffnet, um aus ihr zu lesen, und kein anderer Thread auf die Datei zugreift, sollte die Dateinummer der Datei im Stapel-Frame des Threads und nicht im statischen Speicherbereich abgelegt werden. Gleichermaßen verhält es sich mit Puffern und Arbeitsbereichen, die nur einem Thread dienlich sein sollen; auch sie sollten im Stapel-*Frame* des Threads vorliegen. Auf lokale Stapelvariablen der aktuellen Prozedur kann in den höheren Programmiersprachen und in Assembler einfach zugegriffen werden. Daten, auf die

mehrere Prozeduren zugreifen, können ebenfalls im Stapel abgelegt werden. Pascal-Programme können durch den Datenbereichsmechanismus direkt auf die Daten zugreifen, C- und Assemblerprogramme müssen Zeiger auf die Daten an Prozeduren übergeben.

5.1.3.2 RAM-Semaphoren

Obgleich das Zwischenspeichern lokaler Variablen im Stapel-Frame die zwischen Threads auftretenden Probleme größtenteils reduziert, wird es immer einige Situationen geben, in denen mehr als ein Thread auf statische Daten oder beispielsweise auf eine Dateinummer zugreifen können muß. Dazu sollten die Daten in einem kritischen Bereich gespeichert und RAM-Semaphoren zur Sperrung dieses Bereichs vor der Ausführung eingesetzt werden. Dies stellt sicher, daß zu einem Zeitpunkt immer nur ein Thread auf den kritischen Bereich zugreifen kann (siehe 16.2).

5.1.3.3 DosSuspendThread

In einigen Fällen kann es nötig sein, alle auf einen kritischen Bereich zugreifenden Threads aufzulisten. Dann kann der *DosSuspendThread*-Aufruf zur Unterbrechung von Threads verwendet werden, die zum aufrufenden Prozeß gehören und auf diesen Bereich ebenfalls zugreifen. Zur Unterbrechung eines Threads, der einem anderen Prozeß angehört, kann dieser Aufruf nicht eingesetzt werden. Durch mehrere *DosSuspendThread*-Aufrufe können mehrere Threads (ein Aufruf pro Thread) beendet werden. Falls der gerade unterbrochene Thread einen Systemaufruf ausgeführt hat, ist nicht vorherzusehen, ob der Aufruf im Kernel-Code weiter ausgeführt wird. In jedem Fall gibt es aber keine weitere Ausführung des Anwendungscodes (in Ring 3) durch den abgebrochenen Thread.

Im allgemeinen sollte ein kritischer Bereich durch eine RAM-Semaphore geschützt werden. Die Verwendung einer Semaphore zum Schutz eines kritischen Bereichs entspricht der Verwendung von Ampeln zum Schutz einer Kreuzung (dies ist ein kritischer

Bereich im Straßenverkehr). Die Verwendung von *DosSuspend-Thread* entspricht dem Anhalten der Autos zu dem Zeitpunkt, an dem die Kreuzung von anderen Fahrzeugen benutzt wird. Die Funktionalität der anderen Fahrzeuge wird für den Fall unterbunden, daß diese über dieselbe Kreuzung fahren wollen. Es muß auch sichergestellt sein, daß das anzuhaltende Fahrzeug sich nicht auf der Mitte der Kreuzung befindet, wenn dessen Funktionalität ausgeschaltet wird. Für die Software ist dies ein Thread, der den kritischen Bereich zu dem Zeitpunkt bearbeitet, zu dem die Unterbrechung veranlaßt wird.

Ein *DosResumeThread*-Aufruf stellt die normale Funktionalität aller unterbrochenen Threads wieder her.

5.1.3.4 DosEnterCritSec/DosExitCritSec

Der *DosSuspendThread*-Aufruf unterbricht die Ausführung eines einzigen Threads im Prozeß. *DosEnterCritSec* unterbricht alle Threads eines Prozesses, abgesehen von dem Thread, der den *DosEnterCritSec*-Aufruf initiiert hat. Abgesehen vom Gültigkeitsbereich entsprechen *DosEnterCritSec* und *DosExitCritSec* den Funktionen *DosSuspendThread* und *DosResumeThread*.

Ein *DosExitCritSec*-Aufruf macht einen aktiven *DosSuspend-Thread*-Aufruf rückgängig. Es wird nur die Bearbeitung solcher Threads aufgenommen, die mit *DosEnterCritSec* unterbrochen wurden.

5.1.4 Thread 1

Jeder Thread in einem Prozeß besitzt eine Thread-Kennummer. Die Kennummer des Threads ist für eine Anwendung eine *Black Box*, ihr Wert hat keine Bedeutung. Lediglich das Betriebssystem kennt den dahinter verborgenen Thread; die Thread-Kennummer wird in Betriebssystemaufrufen verwendet. Die einzige Ausnahme bildet der erste Thread eines Prozesses, dessen Nummer immer 1 ist.

Thread 1 verfügt über eine Besonderheit: Er wird unterbrochen, wenn dem Prozeß ein Signal zugeht (siehe Kapitel 12).

5.1.5 Beenden von Threads

Ein Thread kann auf zwei Arten beendet werden: Er kann sich selbst mit einem *DosExit*-Aufruf beenden, oder ein beliebiger Thread im Prozeß kann einen *DosExit*-Aufruf ausführen, bei dem ein Argument zur Beendigung des gesamten Prozesses übergeben wird. Dadurch werden alle zum Prozeß gehörenden Threads sobald wie möglich beendet. Falls ein Thread zum Zeitpunkt eines *DosExit*-Aufrufs gerade einen Teil des Anwendungsprogramms ausführt, wird er sofort beendet. Beim Abarbeiten eines Systemaufrufes wird er baldmöglichst beendet. Wenn der Systemaufruf schnell genug abgearbeitet wird, kann dessen Funktion vervollständigt werden (obwohl die CPU vom Systemaufruf selbst nicht zurückkehrt). Falls der Systemaufruf länger als eine Sekunde andauert, wird er ohne korrekten Abschluß vorzeitig beendet. Dies ist üblicherweise ausreichend, aber in einigen Fällen (beispielsweise beim Schreiben in Einheiten) kann der letzte Schreibvorgang unter Umständen nicht vollständig ausgeführt worden sein.

Zur Beendigung eines Prozesses sollte die *Beenden-des-gesamten-Prozesses*-Form des *DosExit*-Aufrufs statt des *Beenden-des-Threads*-Form verwendet werden.

Dem aufrufenden Prozeß ist nicht bekannt, daß einige *Dynlink*-Pakete (einschließlich einiger OS/2-Systemaufrufe) ihrerseits Threads anlegen. Diese Threads werden *verborgene Threads* (*captive*) genannt, da nur der aufrufende Thread eine Rückmeldung vom *Dynlink*-Aufruf ausgibt. Die davon initiierten Threads bleiben im *Dynlink*-Paket verborgen. Falls ein Programm beendet wird, in dem entsprechende *DosExit*-Aufrufe in der *Beenden-des-Threads*-Form ausgeführt werden, kann die Beendigung unter Umständen nicht gelingen, da für die Anwendung verborgene Threads existieren.

Wird der letzte verbleibende Thread eines Prozesses mit dem *DosExit*-Aufruf in der Form *Beenden-des-Threads* ausgeführt, beendet OS/2 gleichzeitig den gesamten Prozeß.

5.1.6 Durchsatz

Threads wurden entwickelt, um die Ausführungszeiten von Programmen zu verkürzen. In OS/2 Version 1.0 benötigt jeder Thread 1200 Bytes Speicherbereich. Die Beendigung eines Threads gibt diesen Speicher nicht frei, aber nachfolgende Threads verwenden den Speicherbereich beendeter Threads wieder. Der Systemspeicher, den die Threads eines Prozesses benötigen, entspricht der maximalen Anzahl der Threads des Prozesses, die zu einem Zeitpunkt bestehen, multipliziert mit 1200 (Bytes). Diese Speicherkapazität ist den Stapelanforderungen jedes Threads hinzuzurechnen.

Die zur Initiierung eines neuen Threads benötigte Zeit hängt vom Verhalten des vorherigen Threads des Prozesses ab. Die Initiierung eines neuen Threads, der den internen Speicherbereich verwendet, der für einen anderen Thread angelegt wurde, benötigt ungefähr drei Millisekunden. Dies stellt den für den Durchsatz günstigsten Fall dar. Die Anforderung eines neuen Threads, für den kein bereits belegter Speicherplatz verfügbar ist, führt zu einer internen Speicherbelegungsoperation. Diese Operation kann zu einer Speicherverdichtung oder zur Auslagerung eines Segmentes führen, so daß die dafür benötigte Zeit nicht vorhersehbar ist.

Hinweis: Alle Zeitangaben in diesem Buch beziehen sich auf einen 6-MHz-AT mit einem 1-Wait-State-Speicher.

Das System benötigt eine Millisekunde zum Beginn der Ausführung eines nichtblockierten Threads. Mit anderen Worten: Wenn ein Thread niedrigerer Priorität eine RAM-Semaphore freigibt, auf die ein Thread höherer Priorität wartet, vergeht eine Zeitspanne von einer Millisekunde zwischen dem Freigabeaufruf des Threads niedrigerer Priorität und der Rückgabe des *DosSemRequest*-Aufrufs des Threads höherer Priorität.

Threads sind ein Hauptmerkmal von OS/2. Sie werden in künftigen Versionen des Betriebssystem noch umfassender unterstützt werden und in der Architektur eine immer wichtigere Rolle spielen.

5.2 Scheduler/Priorität

Eine typische OS/2-Anwendung arbeitet mit vielen Threads. Häufig sind dabei mehrere Threads zu einem Zeitpunkt zur Ausführung bereit. Der OS/2-Scheduler entscheidet, welcher Thread zuerst ausgeführt wird und wie lange ihm die Rechenzeit bis zur Umschaltung zu einem anderen Thread zugeordnet wird. Der OS/2-Scheduler basiert auf Prioritätsangaben und ist "vorplanend" (*preemptive*). Wenn ein Thread höherer Priorität zur Ausführung bereitsteht, wartet OS/2 nicht, bis der Thread niedrigerer Priorität beendet ist, sondern unterbricht diesen vorzeitig und ordnet die Rechenzeit dem Thread mit der höheren Priorität zu. Der Status des vorzeitig unterbrochenen Threads wird festgehalten, so daß seine Ausführung zu einem späteren Zeitpunkt ohne Nebeneffekte fortgesetzt werden kann.

Der Verteilungsalgorithmus des Schedulers ist geradlinig: Der Thread höchster Priorität wird ausgeführt, solange dieser Rechenzeit benötigt. Bei der Freigabe der CPU durch diesen Thread (beispielsweise durch das Warten auf eine Ein-/Ausgabeoperation) wird dieser Thread nicht länger ausgeführt, und der Scheduler ordnet die CPU-Rechenzeit dem Thread mit der nächstniedrigeren Priorität zu, der zur Ausführung bereitsteht. Wenn ein blockierter Thread wieder ausführbar wird und eine höhere Priorität als der aktuell bearbeitete Thread besitzt, wird die CPU sofort dem ehemals blockierten Thread zugewiesen. Zusammenfassend kann gesagt werden, daß die CPU immer den ausführbaren Thread mit der höchsten Priorität bearbeitet.

Die Verteilung des Schedulers ist einfach: Sie basiert lediglich auf der Überprüfung von Prioritäten. Obwohl OS/2 üblicherweise auf Prozesse konzentriert ist (ein Prozeß wird ausgeführt, beendet, öffnet Dateien usw.), arbeiten die Scheduler-Komponenten wenig mit Prozessen. Hier ist der Thread die Arbeitseinheit, der Scheduler ist in erster Linie Thread-orientiert. Falls Ihnen der Ausdruck *Thread* nicht geläufig ist, können Sie für die folgende Erläuterung den Ausdruck *Thread* gegen *Prozeß* ersetzen. Da die Threads eines Prozesses üblicherweise die gleiche Priorität besitzen, ist die Annahme, daß das System aus

Prozessen besteht, die nach CPU-Ressourcen verlangen, nicht gänzlich falsch.

In OS/2 werden Threads klassifiziert und laufen in einer von drei Kategorien ab: allgemeine Priorität, zeitkritische Priorität und erzwungene Hintergrundpriorität. Die Kategorien werden in Unterkategorien aufgeteilt. In Bild 5.1 wird die Verknüpfung der drei Prioritätskategorien und deren Unterkategorien gezeigt.

5.2.1 Allgemeine Prioritätskategorie

Der Hauptteil der Threads in einem System laufen in der *Kategorie der allgemeinen Priorität* ab und gehören zu einer der drei Unterkategorien *Hintergrund*, *Vordergrund* oder *interaktiv*. Bis zu einer bestimmten Grenze modifiziert OS/2 die Prioritäten der Threads in dieser Kategorie allgemeiner Priorität dynamisch.

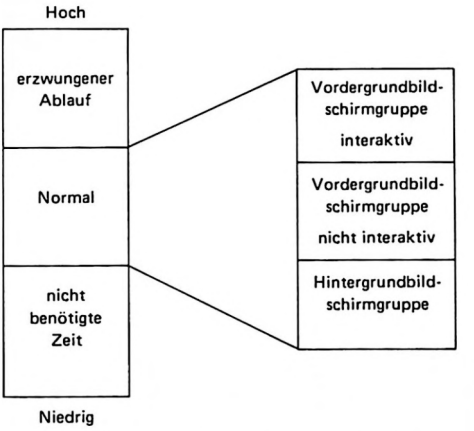


Bild 5.1 Prioritätskategorien

5.2.1.1 Unterkategorie Hintergrund

Der Zweck des OS/2-Prioritätskonzepts ist die Optimierung der Antwortzeiten, nicht des Durchsatzes. Mit anderen Worten: Das System sorgt nicht dafür, daß alle ausführbaren Threads zumindest etwas CPU-Zeit erhalten und achtet auch nicht in erster Linie darauf, daß der Zugriff auf Massenspeichereinheiten optimiert wird, während andere Threads die CPU nutzen. Statt dessen sorgt OS/2 dafür, daß auch weniger wichtige Aufgaben nicht allzu lange auf ihre Ausführung warten müssen. Dies ist der Grund für das Bestehen der Unterkategorie *Hintergrund*. Der Ausdruck *Hintergrund* steht häufig für verschiedene Arten der Vorder- und Hintergrundverarbeitung. Hier wird der Ausdruck für einen Prozeß verwendet, der nicht mit der momentan aktiven Bildschirmgruppe verknüpft ist.

Schaltet ein Benutzer beispielsweise von einem Textverarbeitungsprogramm zu einem im Hintergrund laufenden Tabellenkalkulationsprogramm um, wird das Textverarbeitungsprogramm zu einem Hintergrundprozeß, während das Tabellenkalkulationsprogramm vom Hintergrund in den Vordergrund angehoben wird. Wenn der Benutzer unterschiedliche Bildschirmgruppen wählt, verändert sich der Status von Threads von Vorder- zu Hintergrund und umgekehrt. Hintergrund-Threads haben die niedrigste Priorität in der allgemeinen Prioritätskategorie. Hintergrundapplikationen werden der CPU (und damit der Zugriff auf Disketten/Platten) nur zugewiesen, wenn alle Vordergrund-Threads bearbeitet sind. Sobald ein Vordergrund-Thread ausführbar wird, wird die Rechenzeit dem gerade bearbeiteten Hintergrund-Thread weggenommen und dem Vordergrund-Thread zugeteilt. Hintergrund-Threads wird die nicht benutzte Systemzeit zugeteilt, sie können nicht mit Vordergrund-Threads konkurrieren.

5.2.1.2 Unterkategorien Vordergrund und interaktiv

Alle mit der aktiven Bildschirmgruppe verbundenen Prozesse gehören zur Unterkategorie *Vordergrund*; der aktuell mit der Tastatur arbeitende Prozeß wird in die Unterkategorie *interaktiv* eingeordnet. Dies stellt sicher, daß der Benutzer die schnellstmögliche Antwort auf einen Befehl erhält. Wenn alle Threads des interaktiven Prozesses die CPU freigeben (aufgrund eines blockierenden Systemaufrufes), erhält ein Vordergrund-Thread, der nicht interaktiv ist, den nächsten CPU-Zeitabschnitt, da diese Threads üblicherweise Aufgaben erledigen, die mit dem interaktiven Prozeß verknüpft sind. Falls kein Vordergund-Thread die CPU-Zeit benötigt, kann ein Hintergrund-Thread ausgeführt werden.

Obwohl sich der Scheduler mit Threads und nicht mit Prozessen beschäftigt, wird der Status der Prozesse zwischen den Kategorien - Vordergrund, Hintergrund und interaktiv - verteilt. Wenn die Kategorie eines Prozesses sich ändert - beispielsweise benötigt ein Prozeß die Zuordnung in die interaktive Kategorie, da Tastatureingaben ausgeführt werden müssen - werden die Prioritäten aller Threads entsprechend angepaßt.

Es scheint so, als ob bei diesem Prinzip einige Hintergrund-Threads mit niedriger Priorität niemals ausgeführt werden. Dies ist nicht der Fall, wenn der Systemzustand über einen längeren Zeitraum betrachtet wird. In einem 5-Sekunden-Intervall kann es sehr wohl vorkommen, daß einem Hintergrund-Thread keine Ausführungszeit zugeordnet wird, über eine Minute gesehen wird dies aber nur in Ausnahmefällen vorkommen. Interaktive Befehle benötigen selten mehr als einige wenige Sekunden Rechenzeit. Befehle für die Datenübertragung auf Disketten oder Platten können zwar einen längeren Zeitraum benötigen, während des Wartens auf die Ausführung der Disketten-/Plattenoperation benötigen sie aber keine Rechenzeit. Hinzu kommt, daß der Benutzer einer interaktiven Anwendung ein System nicht immer beschäftigt hält. Das Eingeben und Überlegen zwischen den einzelnen Kommandos läßt genügend Zeit zur Ausführung von Hintergrund-Threads.

Wie ist dies mit dem *Presentation Manager* vereinbar? Der Presentation Manager führt mehrere unabhängige, interaktive Tasks in derselben Bildschirmgruppe aus - sind das alles Vordergrund-Threads? Wie erkennt OS/2, welches der interaktive Prozeß ist? Die Antwort besteht darin, daß der Presentation Manager Anweisung an den Scheduler übergibt. Wenn die Bildschirmgruppe des Presentation Manager angezeigt wird, so werden alle Threads der Bildschirmgruppe in die Kategorie Vordergrund angehoben. Wählt der Benutzer ein bestimmtes Fenster zum Empfang von Tastatur- oder Mauseingaben, teilt der Presentation Manager dem Scheduler mit, daß dieser Prozeß nun der interaktive Prozeß ist. Das Ergebnis ist, daß der interaktive Durchsatz des Systems auch in der Bildschirmgruppe des Presentation Manager erhalten bleibt.

5.2.1.3 CPU- oder Ein-/Ausgabebindung

Es ist bekannt, daß einige Betriebssysteme einen maximalen Systemdurchsatz durch die gleichzeitige Ausführung CPU- und ein-/ausgabegebundener Anwendungen zu realisieren versuchen. In der Theorie sieht das so aus, daß das ein-/ausgabegebundenes Anwendungsprogramm zwar die Laufwerke anspricht, aber nur wenig CPU-Zeit benötigt. Die Übertragung von Daten von und zu Disketten/Platten kann erledigt werden, während ein CPU-gebundenes Task von der CPU abgearbeitet wird. Falls der ein-/ausgabegebundene Thread die höhere Priorität besitzt, laufen beide Tasks gleichzeitig ab. Sobald eine Disketten-/Plattenoperation beendet ist, wird dem ein-/ausgabegebundenen Thread die CPU-Rechenzeit zugewiesen und er initiiert die Ausführung einer weiteren Disketten-/Plattenoperation. Die übrige CPU-Zeit wird der CPU-gebundenen Aufgabe zugewiesen, die in diesem Fall eine niedrigere Priorität besitzen muß.

Diese Methode funktioniert nicht, wenn der CPU-gebundene Thread eine höhere Priorität als der ein-/ausgabegebundene Thread besitzt. Der CPU-gebundene Thread wird die CPU für sich beanspruchen und der ein-/ausgabegebundene Thread wird überhaupt keine CPU-Zeit zur Veranlassung weiterer Ein-/Ausgabeanforderungen erhalten. Die üblichen Scheduler lösen

dieses Problem, indem die Priorität des ein-/ausgabegebundenen Threads erhöht und die Priorität der CPU-gebundenen Aufgabe vorläufig erniedrigt wird. Der ein-/ausgabegebundene Thread erhält dadurch genügend Systemzeit, um Ein-/Ausgabeanforderungen initiieren zu können.

Auch der OS/2-Scheduler basiert zu einem gewissen Grad auf diesem Konzept. Bei jedem blockierenden Systemaufruf achtet der Scheduler auf die vergangene Zeit zwischen dem Zeitpunkt, zu dem die CPU-Zeit dem Thread zugeordnet wurde, und dem Zeitpunkt, zu dem der Thread sich selbst mit einem Systemaufruf blockiert. In diesem Zusammenhang wird der Ausdruck *blockieren* für die Anforderung einer Ein-/Ausgabeoperation verwendet. Ist das Zeitintervall kurz, wird der Thread als ein-/ausgabegebunden angesehen und seine Priorität um einen kleine Betrag erhöht. Falls der Thread tatsächlich ein-/ausgabegebunden ist, wird er mehrmals erhöht und erhält so eine gewichtige Prioritätssteigerung. Benötigt ein Thread die CPU für eine längere Periode, wird er als CPU-gebunden angesehen und seine Priorität um einen kleinen Betrag erniedrigt.

Hinweis: Nahezu alle blockierenden Operationen warten auf eine Ein- oder Ausgabe. Sind die Daten eines Threads im Puffer, kann der Thread viele Ein-/Ausgabeanforderungen initiieren und ist dennoch CPU-gebunden. Mit anderen Worten: Wenn hier von ein-/ausgabegebundenen Threads gesprochen wird, sind einheitengebundene Operationen und nicht unbedingt nur ein-/ausgabegebundene Operationen gemeint.

Die Steigerung der Priorität für ein-/ausgabegebundene Threads ist nur gering. Kein Hintergrund-Thread, gleich wie ein-/ausgabegebunden er auch sein mag, kann seine Priorität bis zu einem Punkt erhöhen, an dem er eine höhere Priorität als ein Vordergrund-Thread besitzt (auch wenn dieser CPU-gebunden ist). Die Durchsatzoptimierung bezieht sich nur auf Threads, die eine gleiche Priorität besitzen. Die Threads eines Prozesses besitzen meist die gleiche Basispriorität, so daß die Feinanpassungen den Durchsatz von Prozeßteilen beschleunigen.

5.2.2 Kategorie Zeitkritische Priorität

Vordergrund-Threads und dabei insbesondere interaktiven Vordergrund-Threads wird die CPU-Zeit zu einem beliebigen Zeitpunkt zugewiesen. Nicht-interaktive Vordergrund-Threads und Hintergrund-Threads können CPU-Zeit nicht beliebig erlangen (das ist immer nur für eine willkürliche Zeitspanne wahr). Zur Verbesserung der Antwortzeiten ist diese Methode optimal, bei der Verwendung von Netzwerk- oder Telekommunikationsanwendungen kann die Verbindung mit anderen Systemen aber verlorengehen, wenn ankommende Datenpakete nicht in einem gewissen Zeitrahmen beantwortet werden. Es könnte auch sein, daß man vom Prinzip der *optimalen Antwortzeiten* abweichen will, wenn es um Druckvorgänge geht. Die meisten Drucker sind sehr langsam und Spooler-Programme benötigen häufig nur wenig CPU-Zeit. Aus diesem Grund läuft der OS/2-Drucker-Spooler (Programm zur Verwaltung der Druckerwarteschlange) mit einer sehr hohen Priorität, um den Drucker ständig beschäftigt zu halten.

Anwendungen werden zeitkritisch genannt, wenn die Ausführung einer ihrer Aufgaben in einem bestimmten Zeitrahmen erfolgen muß. Zeitkritische Anwendungen können (müssen aber nicht) interaktiv sein, sie können in einer Vorder- oder Hintergrundbildschirmgruppe ablaufen, dies alles sollte aber nicht ihre hohe Priorität beeinflussen. Zur Berücksichtigung zeitkritischer Anwendungen verfügt der OS/2-Scheduler über die Kategorie der *zeitkritischen Priorität*. Ein in dieser Prioritätskategorie ausgeführter Thread wird mit höherer Priorität als ein Thread, der nicht zeitkritisch ist, (einschließlich interaktiver Threads) bearbeitet. Im Gegensatz zu Prioritäten der allgemeinen Kategorie wird eine zeitkritische Priorität unter keinen Umständen angepaßt. Es erfolgt lediglich die einmalige Vergabe des Prioritätsstatus, der unverändert bleibt, bis ein Systemaufruf Einfluß darauf nimmt.

Natürlich sollten zeitkritische Threads nur einen Teil der CPU-Zeit für sich beanspruchen. Eine Anwendung mit einem zeitkritischen Thread, der viel CPU-Zeit für sich in Anspruch nimmt

(beispielsweise 20 Prozent), verlangsamt interaktive Vordergrundanwendungen oder bringt diese sogar zeitweilig (für den Benutzer erkennbar) zum erliegen. Die Nutzbarkeit des Systems wird negativ beinflußt, wenn eine interaktive Anwendung nicht sofort bearbeitet wird. Die Bildschirmausgabe verlangsamt sich und wird teilweise sogar unterbrochen.

Nicht alle Threads eines Prozesses müssen die gleiche Priorität besitzen. So kann eine Anwendung eine zeitkritsche Antwort für einen Teil ihrer Aufgaben benötigen, während die anderen Aufgaben mit normaler Priorität ausgeführt werden können. In einem Datenfernübertragungsprogramm kann der Empfang ankommender Daten mit zeitkritischer Priorität ausgeführt werden, die Einreihung von Meldungen in Warteschlangen ist dagegen sicherlich eine Aufgabe für einen Thread mit normaler Priorität. Falls der zeitkritische Thread feststellt, daß der Thread normaler Priorität seine Aufgabe nicht zeitgerecht bewältigen kann, wird eine Meldung (*Warte mit der weiteren Sendung*) an das sendende Programm übertragen.

Es wird empfohlen, daß Prozesse, die für die Überwachung zuständig sind, nur den Monitor-Thread mit einer zeitkritischen Priorität ausführen. Verzögerte Einheitenantworten entfallen, die aufgrund einer zu langsam ausgeführten Monitordatenbearbeitung entstehen könnten (siehe 16.1).

5.2.3 Kategorie Niedrige Priorität

Wenn man sich die *allgemeine Prioritätskategorie* als Bereich von Prioritätsebenen vorstellt, ist die *zeitkritische Prioritätskategorie* am oberen Ende anzusiedeln, während die *niedrige Hintergrundpriorität* am unteren Ende erscheint. Die Hintergrundpriorität stellt das Gegenteil zur zeitkritischen Prioritätskategorie dar. Die Threads in der Hintergrundpriorität erhalten CPU-Dienste nur dann, wenn keine Vordergrund-Threads zur Ausführung bereitstehen. Der Systemaufruf zum Anlegen eines Threads setzt gleichzeitig dessen Priorität, die von OS/2 nicht verändert wird.

Diese Prioritätsebene wird im allgemeinen nicht stark frequentiert, sie kann aber für weniger wichtige Aufgaben, wie dem Aufzeichnen der Rechenzeitverteilung, verwendet werden.

5.2.4 Setzen von Prozeß-/Thread-Prioritäten

In diesem Abschnitt wurden die verschiedenen Prioritäten und ihre Auswirkungen auf das Gesamtsystem erläutert. Wie die Prioritäten gesetzt werden, wurde bislang verschwiegen. Da die Vererbung ein wichtiges OS/2-Konzept ist, stellt sich die Frage, wie die Priorität eines Vaterprozesses die des Kindprozesses beeinflußt.

Beim Anlegen eines Threads wird die Priorität des aufrufenden Threads vererbt. Im Falle des *DosCreateThread*-Aufrufes ist der aufrufende Thread der Anlage-Thread. Für den Thread 1 ist der Thread des Vaterprozesses, der den *DosExecPgm*-Aufruf ausführte, der Anlage-Thread. Bei der Ausführung eines *DosSetPrty*-Aufrufes tritt die neue Prioritätsebene sofort in Kraft. Bei der Prioritätsänderung der Threads eines anderen Prozesses gilt die Änderung nur für die Threads, deren Priorität nicht explizit vom eigenen Prozeß festgelegt wurden. Ein Vaterprozeß kann auf diese Weise nicht versehentlich die Priorität eines zeitkritischen Threads ändern.

In Zukunft wird der Algorithmus dahingehend verbessert werden, daß jeder Thread eine eigene Basispriorität besitzt. Ein neuer Thread wird die Basispriorität des anlegenden Threads erben. Eine Änderung der Thread-Priorität eines Prozesses wird dafür sorgen, daß die Priorität aller Threads um den gleichen Betrag angehoben oder gesenkt wird. Durch die Veränderung der Basispriorität werden Threads in der zeitkritischen oder niedrigen Prioritätskategorie nicht beeinflußt werden.

Kapitel 6

Die Benutzerschnittstelle

In OS/2 sind Untersysteme implementiert, die in weiteren Versionen problemlos austauschbar sind: Das Dateiverwaltungssystem, die Speicherverwaltung, das Multitasking-Untersystem und die Benutzerschnittstelle (den Presentation Manager). Das Betriebssystem MS-DOS definiert und unterstützt kein besonderes Benutzerschnittstellensystem, das speziell für die Betriebssystembenutzung entwickelt wurde. Deswegen muß jede Anwendung eine eigene Benutzerschnittstelle bereitstellen. MS-DOS-Utilities besitzen eine einfache zeilenorientierte Schnittstelle, die sich im wesentlichen nicht von der vom System bereitgestellten TTY-Schnittstelle unterscheidet.

Demgegenüber ist OS/2 ein graphikorientiert arbeitendes Betriebssystem und benötigt auf jeden Fall eine standardisierte graphische Benutzerschnittstelle (GUI für *graphical user interface*). Da die Softwareentwicklung für solche Systeme sehr zeitaufwendig, kompliziert und anspruchsvoll ist, kann der Betriebssystemlieferant nicht voraussetzen, daß jedes Softwarehaus und jeder Programmierer einer einzelnen Anwendung, eine eigene graphikorientierte Benutzerschnittstelle anbietet. Ein Hauptvorteil einer definierten graphischen Benutzerschnittstelle liegt darin, daß die Anwendungsprogramme eine identische Oberflächendarstellungsform wie auch Dateitransfermöglichkeiten anbieten, also Daten aus verschiedenen Programmen miteinander vermischt werden können. So können verschiedene Anwendungsprogramme eine identische Bildschirmdarstellung (mehrere Ausgabefenster) haben, sich Daten teilen und der Benutzer kann problemlos Daten zwischen den Anwendungen hin

und her übertragen. Falls jede Anwendung ein eigenes GUI-Paket besitzen würde, wäre eine gemeinsame Datennutzung unmöglich. Die graphische Benutzerschnittstelle erleichtert zudem die Systemanwendung für den Anwender: Da für alle Anwendungsprogramme eine vorgegebene, definierte Benutzeroberfläche zur Verfügung gestellt wird, ist ein Springen von Anwendungsprogramm zu Anwendungsprogramm ohne große Anpassungsschwierigkeiten in der Bedienung möglich.

6.1 VIO-Benutzerschnittstelle

Das vollständige graphische Benutzerschnittstellenuntersystem wird in OS/2 Version 1.0 nicht implementiert sein; die erste Version besitzt nur ein zeichenorientiertes VIO/KBD/MOU-Untersystem (Kapitel 13). Das VIO-Paket stellt keine graphikorientierten Dienstleistungen bereit; es ist Anwendungen möglich, das VIO zu umgehen und eigene Routinen zu erstellen. Beim Systemstart wird auf dem Bildschirm eine Liste der verfügbaren Bildschirmgruppen aufgelistet. der Benutzer kann eine Bildschirmgruppe wählen oder eine neue anlegen. Aus dieser Bildschirmgruppe heraus bringt die Betätigung einer definierten Tastenfolge die Liste der Bildschirmgruppen wieder auf den Bildschirm. Eine andere Tastenfolge erlaubt dem Benutzer die Umschaltung zwischen existierenden Bildschirmgruppen. Eine der Bildschirmgruppen wird in der Liste als Real-Modus-Bildschirmgruppe identifiziert.

6.2 Benutzerschnittstelle des Presentation Manager

Der OS/2-Presentation Manager ist eine leistungsfähige, graphische Benutzerschnittstelle. Leistungsmerkmale wie Fenster, *Drop-Down*-, *Pop-Up*-Menüs und Rollbalken werden unterstützt. Mit einem Zeigeinstrument wie der Maus ist die Bedienung sehr komfortabel, aber auch über die Tastatur ist eine uneingeschränkte Steuerung möglich.

Der Presentation Manager besitzt Bildschirmfenster, die mehreren Anwendungen die gleichzeitige Verwendung des Bildschirms (und der Tastatur) ermöglichen. Jede Anwendung verwendet ein oder mehrere Fenster zur Datenausgabe; der Benutzer kann die Fenster an unterschiedlichen Stellen auf dem Bildschirm positionieren und deren Größe verändern. Fenster können zu *Ikonen* (Symbolen) zusammenschrumpfen und dürfen sich überlappen. Mit der Maus oder Tastatur kann bei der Eingabe zwischen den Fenstern gesprungen werden. Der Presentation Manager leitet Tastenanschläge und Mausknopfbetätigungen automatisch an die korrekte Anwendung um.

Durch die Fenstertechnik muß der Presentation Manager nicht den OS/2-Bildschirmgruppenmechanismus benutzen, um den Benutzer zwischen verschiedenen Anwendungen umschalten zu lassen. Der Benutzer startet eine Anwendung durch Zeigen mit dem Mauszeiger auf den Programmnamen in der Menüanzeige; für die meisten Anwendungen legt der Presentation Manager ein neues Fenster an und weist diesem einen Prozeß zu. Einige Anwendungen können die graphische Benutzerschnittstelle des Presentation Manager umgehen und die Anzeige direkt kontrollieren. Bei der Ausführung einer solchen Anwendung legt der Presentation Manager dafür eine eigene Bildschirmgruppe an und aktiviert diese. Der Benutzer kann ein Menü anzeigen lassen und daraus jedes ablaufende Programm anwählen. Falls das gewählte Programm die Standard-Presentation-Manager-GUI verwendet, wird in die von diesem Programm verwendete Bildschirmgruppe umgeschaltet. Andernfalls wird die der Anwendung zugeordnete Bildschirmgruppe aktiviert.

Zusammenfassend kann gesagt werden, daß nur *Real*-Modusanwendungen und Anwendungen, die die direkte Steuerung des Bildschirms wünschen, in eigenen Bildschirmgruppen ablaufen müssen. Der Presentation Manager verwaltet alle anderen Prozesse in einer einzigen Bildschirmgruppe und verwendet die Fenstertechnik zur gemeinsamen Benutzung eines Bildschirmes. Der Benutzer kann über ein Menü zwischen den Anwendungen wählen. Verwenden zwei Anwendungen die Standardschnittstel-

le, so kann der Benutzer ohne Menüaufruf zwischen beiden umschalten.

6.3 Presentation Manager und VIO-Kompatibilität

Unter OS/2 Version 1.1 und allen folgenden Versionen wird der Presentation Manager die VIO-Schnittstelle ersetzen. Anwendungen, die die zeichenorientierte VIO-Schnittstelle verwenden, werden auch unter dem Presentation Manager korrekt ausgeführt werden. Gleiches gilt für Anwendungen, die STDIN und STDOUT-Dateinummern zur interaktiven Ein-/Ausgabe verwenden. Software, die die VIO-Schnittstelle zum direkten Zugriff auf den Bildschirm einsetzt, wird ebenfalls unterstützt. Wie bereits beschrieben, kann der Presentation Manager solche Anwendungen in einer Bildschirmgruppe ausführen, die nur dieser Anwendung zugeordnet sind.

Kapitel 7

Dynamische Verknüpfung

Eine zentrale Komponente von OS/2 ist die dynamische Verknüpfung (*Dynamic Link* oder kurz *Dynlink*). Die dynamische Verknüpfung spielt bei mehreren Aspekten der Betriebssystemarchitektur eine entscheidende Rolle. Zunächst soll ihre Arbeitsweise erläutert werden.

7.1 Statische Verknüpfung

Bevor man sich mit der dynamischen Verknüpfung beschäftigt, sollte man wissen, was eine statische Verknüpfung ist. Im Grunde ist jeder Programmierer, der mehr kennt als die Interpreter-Sprache BASIC, mit dem Prinzip der statischen Verknüpfung vertraut. Ein Beispiel: In einem Programm wird eine Unterroutine oder eine Prozedur aufgerufen, die im Quellcode des Programms nicht vorhanden ist. Die Routine bzw. Prozedur muß als *extern* deklariert werden, so daß der Assembler oder Compiler den Aufruf als nicht definiertes Symbol behandeln kann. Beim Binden (Linken) ist dem Linker neben den aus dem Quellcode erzeugten .OBJ-Dateien eine weitere .OBJ-Datei (oder eine Bibliothek mit der .OBJ-Datei) zu übergeben, die den Code der aufgerufenen Routine enhält. Der Linker verknüpft die Objektdateien in ein ausführbares Programm - die .EXE-Datei - die sowohl die externe Routine als auch die aufrufenden Routinen enthält. Während des Verknüpfungsprozesses paßt der Linker die Aufrufe an externe Funktionen, die bis dahin lediglich nicht definierte Bezüge waren, an die Struktur der

.EXE-Datei an, so daß die Aufrufe auf die entsprechenden Einsprungspunkte der Routinen und Prozeduren erfolgen. Dieser Vorgang wird in Bild 7.1 gezeigt.

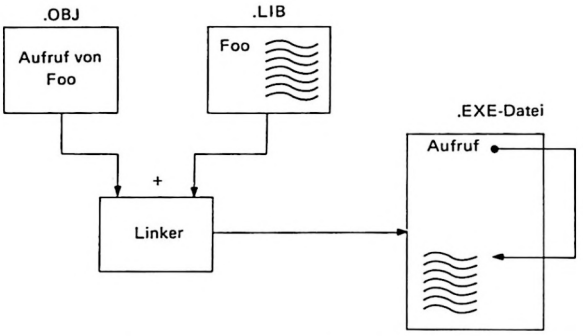


Bild 7.1 Statische Verknüpfung

Durch das Leistungsmerkmal der statischen Verknüpfung kann ein Programmsystem in mehrere Module gesplittet und damit übersichtlich gehalten werden. Die einzelnen Teile (Module) können einzeln kompiliert werden, indem die in einer Datei nicht enthaltenen Aufrufreferenzen als *extern* deklariert werden. Der Linker verknüpft die kompilierten Teilmodule zu einer .EXE-Datei, wobei die externen Referenzen (das sind Referenzen zwischen verschiedenen Programmodulen) aufgelöst werden.

Die Aufteilung eines Programms in mehrere Module erleichtert die Entwicklung und Wartung ungemein. Der Hauptvorteil der statischen Verknüpfung liegt allerdings darüber hinausgehend in der Möglichkeit, auf Bibliotheken mit Standardroutinen zugreifen zu können, ohne sich mit dem Quellcode der Routinen beschäftigen zu müssen. Der Quelltext muß nicht kompiliert werden, man muß ihn zur Ausführung der Routinen nicht einmal besitzen. Nahezu alle Compiler für eine höhere Programmiersprache werden mit einer oder mehreren Standardlaufzeitbibliotheken ausgeliefert, die Unterroutinen enthält, die der Compiler implizit oder der Programmierer explizit aufrufen kann. Die

Quelldateien der Laufzeitbibliotheken werden nur selten mit dem Endprodukt vertrieben. Auf den Programmdisketten sind meist nur die .OBJ-Dateien (Objektdateien) in einer Bibliothek enthalten.

Zusammenfassend kann man sagen, daß beim traditionellen statischen Verknüpfen die einzelnen Programmmodule (das sind auch die externen Unterroutinen) beim Linken vorhanden sein müssen, um in eine endgültige .EXE-Datei eingebunden werden zu können. Die .EXE-Datei wird durch das Einbinden umfangreicher. Dies ist aber nur ein kleiner Nachteil. Gravierender ist, daß der Code eines Programmmoduls nicht verändert werden kann, ohne die gesamte Datei nochmals linken zu müssen. Da PC-Software aus kommerziellen Gründen überwiegend als ausführbares Programm geliefert wird (und nicht als Quell- oder Objektcode), liegt das Kompilieren und Linken im allgemeinen außerhalb des Einflußbereiches des Endbenutzers. Die Folge: Das Zielprogramm kann nicht von mehreren (verschiedenen) Anwendungen, die die gleichen Bibliotheksroutinen einsetzen, verwendet werden. Dies hat zwei Gründe: Zum ersten wird das Zielprogramm für jedes Programm vom Linker unterschiedlich zusammengebaut; auch wenn das Programm für jede Anwendung logisch denselben Inhalt besitzt, sind die binären Instruktionen in jeder .EXE-Datei unterschiedlich. Zum zweiten hat das Betriebssystem keine Kenntnis davon, daß unterschiedliche Anwendungen die gleiche Bibliothek benutzen. Für das Betriebssystem ist nicht ersichtlich, an welchen Stellen in den einzelnen .EXE-Dateien die Bibliothek jeweils steht. Daher ist es unvermeidlich, die Bibliothek mehrmals im Hauptspeicher des Rechners zu halten.

7.2 Dynamische Verknüpfung beim Laden

Der mechanische Prozeß bei der *dynamischen Verknüpfung beim Laden* (*loadtime dynamic linking*) entspricht dem der statischen Verknüpfung. Der Programmierer gibt eine externe Referenz auf eine Unterroutine an und spezifiziert beim Linken eine Bibliotheksdatei (oder eine .OBJ-Datei), in der die Referenz defi-

Technische Hochschule Zittau
Hochschulbibliothek

niert wird. Der Linker erzeugt eine .EXE-Datei, die von OS/2 geladen und ausgeführt werden kann. Genauer betrachtet sind die Vorgänge bei statischer und bei dynamischer Verknüpfung allerdings vollkommen unterschiedlich.

Der erste Schritt ist bei beiden Arten der Verknüpfung identisch. Die externe Referenz wird kompiliert oder assembliert, so daß eine .OBJ-Datei mit einem Datensatz für jede externe Referenz entsteht. Der Assembler oder Compiler nimmt auf die dynamische Verknüpfung keine Rücksicht. Die vom Assembler oder Compiler erzeugte .OBJ-Datei kann sowohl zur statischen als auch zur dynamischen Verknüpfung eingesetzt werden (oder, was häufig vorkommt, zur Kombination beider Verknüpfungsarten: einige externe Referenzen werden dynamisch verknüpft, andere statisch).

Bei der statischen Verknüpfung findet der Linker die aktuelle externe Referenz in der Bibliotheksdatei. Bei der dynamischen Verknüpfung findet der Linker einen Datensatz, der eine Modulnamenzeichenkette und einen Einsprungspunkt definiert. Ein Beispiel: Eine Routine trägt den Namen FOO. Die Bibliotheksdatei kann nun nur diese Namenzeichenkette und nicht den Programmcode von FOO selbst enthalten. Der Einsprungsnamensstring muß nicht der Name sein, durch die das Programm die Routine aufrief. Die resultierende .EXE-Datei enthält nicht den Programmcode von FOO, sondern lediglich einen Datensatz zur dynamischen Verknüpfung, der dieses Modul und den Einsprungspunktnamen für FOO spezifiziert. Der Prozeß ist in Bild 7.2 gezeigt.

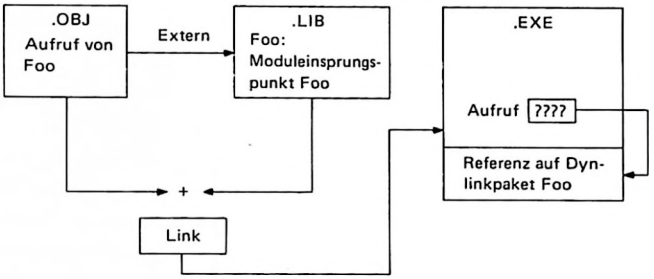


Bild 7.2 Dynamische Verknüpfung

Beim Ausführen der .EXE-Datei lädt OS/2 den .EXE-Programmcode in den Speicher und entdeckt dabei die Datensätze zur dynamischen Verknüpfung. Für jedes aufgeführte dynamische Modul sucht OS/2 den Programmcode im Bibliotheksverzeichnis zur dynamischen Verknüpfung und lädt ihn in den Speicher (sofern das Modul nicht schon benutzt ist). Das System verknüpft die externen Referenzen der Anwendung mit der Adresse des aufgerufenen Einsprungspunktes. Dieser Prozeß wird in Bild 7.3 gezeigt.

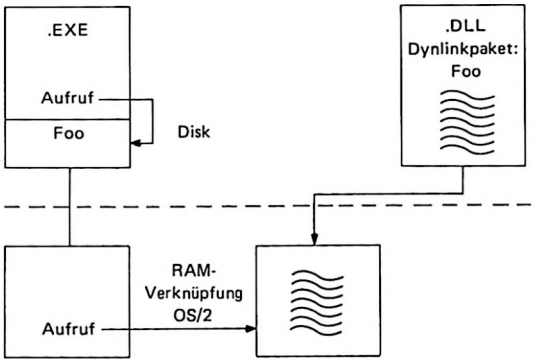


Bild 7.3 Erstellung der dynamische Verknüpfung beim Laden

Zusammenfassend kann gesagt werden, daß der Zielcode beim Binden in die Quelldatei einbezogen wird. Der Linker schreibt den Modulnamen und einen Einsprungspunkt in die .EXE-Datei. Beim Laden des Programms (das heißt bei der Ausführung) sucht OS/2 den Zielcode, lädt diesen ebenfalls und führt die nötigen Verknüpfungen aus. Obwohl die Verknüpfung im Vergleich zum statischen Vorgang beim ladedynamischen nur auf den späteren Zeitpunkt beim Laden verschoben wird, besitzt diese Methode mehrere signifikante Vorteile. Zum ersten liegt der Zielcode nicht in der .EXE-Datei, sondern in einer eigenständigen dynamischen Link-Bibliothek (mit der Namenserweiterung DLL für *Dynamic Link Library*) vor. Aus diesem Grund ist die .EXE-Datei kleiner, da sie nur den Namen des Zielcodes, nicht aber den Code selbst enthält. Bei der Veränderung oder Aktualisierung des Zielcodes wird einfach die .DLL-Datei ersetzt. Beim nächsten Laden der Anwendungsdatei (mit einigen Einschränkungen, siehe Punkt 7.11.2) wird die neue Version des Zielcodes mit dem Anwendungsprogramm verknüpft. Das Vorliegen des Zielcodes in einer eigenständigen .DLL-Datei ebnet den Weg für die gemeinsame Benutzung eines Programmes. OS/2 erkennt, daß zwei Anwendungen denselben Dynlink-Code ansprechen, da die Dynlink-Routine von OS/2 geladen und verknüpft wurde. Diese Kenntnis kann zur gemeinsamen Benutzung von Segmenten des Dynlink-Paketes statt des Ladens von Zweitkopien verwendet werden.

Ein weiterer Vorteil der dynamischen Verknüpfung liegt schließlich darin, daß der Verknüpfungsvorgang für den Benutzer nicht erkennbar ist (eventuell läßt sich dies sogar für den Programmierer erreichen). Die Technik der dynamischen Verknüpfung muß nur verstanden werden, wenn ein solches Modul entwickelt werden soll. Die Benutzung des Moduls ist sogar möglich, wenn nicht einmal Kenntnisse über die statische Verknüpfung vorhanden sind. Der einzige Nachteil des dynamischen Linkens besteht darin, daß das Laden eines Programms teilweise etwas länger dauert. Da der Zielcode zur dynamischen Verknüpfung in einer von der .EXE-Datei verschiedenen Datei

vorliegt, müssen einige weitere Disketten-/Plattenoperationen ausgeführt werden.

Der Geschwindigkeitsverlust hängt von der Art des Dynlink-Moduls sowie davon ab, ob die .EXE-Datei die erste Datei ist, die sich auf dieses Modul bezieht (siehe 7.11 Implementationsdetails).

Bisher wurden nur Prozesse erwähnt, die Dynlink-Routinen aufrufen. Dynlink-Routinen können natürlich auch von anderen Dynlink-Routinen angesprochen werden. Beim Laden einer Dynlink-Routine als Antwort auf eine Prozeßanfrage untersucht OS/2 die Routine auf weitere Dynlink-Referenzen. Anschließend werden alle Referenzen durch das Laden der entsprechenden Dynlink-Routinen erfüllt.

7.3 Dynamische Verknüpfung zur Laufzeit

Die bisher beschriebene Verknüpfung wird beim Ladevorgang ausgeführt. Dabei müssen nicht alle dynamischen Verknüpfungen beim Laden in der .EXE-Datei gefunden werden. Ein Dynlink-Paket kann auch während der Ausführung eines Programms hinzugefügt werden. Dies geschieht in gleicher Weise wie bei der ladedynamischen Verknüpfung, abgesehen davon, daß der Prozeß den Namen des Dynlink-Moduls und den Einsprungspunkt anlegt und diese anschließend an OS/2 übergibt, so daß das Betriebssystem den spezifizierten Dynlink-Code laden kann.

Die Verknüpfung zur Laufzeit erfolgt in vier Schritten:

1. Der Prozeß führt einen *DosLoadModule*-Aufruf aus, mit dem OS/2 der Name des zu ladenden Dynlink-Codes mitgeteilt wird.

2. Ein *DosGetProcAddr*-Aufruf dient dem Erhalt der Adressen der Routinen, auf die der Prozeß zugreifen soll.

3. Der Prozeß spricht die Dynlink-Bibliothekseinsprungspunkte durch einen indirekten Aufruf über die vom *DosGetProcAddr*-Aufruf gelieferte Adresse an.

4. Wenn der Prozeß keine weitere Verwendung für den Dynlink-Code hat, kann er *DosFreeModule* zur Freigabe des Dynlink-Codes aufrufen. Danach verfügt der Prozeß noch über die von *DosGetProcAddr* gelieferte Adresse, die aber nun ungültig ist. Eine Referenz auf diese Adresse führt zu einem GP-Fehler.

Die dynamische Verknüpfung zur Laufzeit sollte eingesetzt werden, wenn ein Programm Module anspricht, von denen zum Ladezeitpunkt nicht bekannt ist, ob sie benötigt werden. So könnte zum Beispiel ein Zeichenprogramm vier Plotter unterstützen, wobei die Treiber als Dynlink-Pakete implementiert sind. Es ist nicht sinnvoll, alle Treiber beim Laden einzubeziehen, da nur einer genutzt werden wird und die anderen lediglich Speicher- und Auslagerungskapazität beanspruchen. Statt dessen kann das Zeichenprogramm auf die Eingabe eines Befehls warten, der den anzusprechenden Plotter definiert. Anschließend wird das Leistungsmerkmal der laufzeitdynamischen Verknüpfung zum Laden des entsprechenden Paketes genutzt. Die Anwendung muß auch bei der Initialisierung noch keinen *DosLoadModule*-Aufruf ausführen. Dies hat Zeit, bis der Benutzer einen Plotterbefehl ausführt.

Eine Anwendung muß nicht einmal alle Module und Einsprungspunkte kennen, die aufgerufen werden. Vielmehr können die Namen der Dynlink-Module durch andere Prozesse oder etwa durch eine Benutzereingabe spezifiziert werden. Möglich ist auch der Rückgriff auf eine Konfigurationsdatei. Dem Benutzer des Zeichenprogramms kann somit die Installation von weiteren Plotter-Treibern ermöglicht werden, die bei der Entwicklung der Graphikanwendung noch nicht existierten. In diesem Beispiel müssen die Aufrufsequenzen der Dynlink-Plotter-Treiber standardisiert sein, und das Programm muß selbstverständlich über eine Möglichkeit verfügen, die neuen Routinen korrekt anzusprechen.

Natürlich ist dieser Prozeß nicht auf die Verwendung eines einzigen Dynlink-Moduls beschränkt. Mehrere Aufrufe von *DosLoadModule* können zur Verknüpfung mehrerer Dynlink-Mo-

dule herangezogen werden. Unabhängig von der Anzahl der verwendeten Module sollte ein *DosFreeModule*-Aufruf erfolgen, wenn ein Dynlink-Modul nicht mehr benötigt und der Prozeß weiterhin ausgeführt wird. Die Durchführung von *DosFreeModule*-Aufrufen kann entfallen, wenn der Prozeß beendet wird. In diesem Fall gibt OS/2 alle Dynlink-Module bei der Prozeßbeendigung frei.

7.4 Dynamische Verknüpfung, Prozesse und Threads

Vereinfacht gesagt, behandelt OS/2 die dynamischen Verknüpfungen als Referenzen auf Unterroutinenpakete. Dynamische Verknüpfungen sind keine Prozesse und verfügen nicht über Ressourcen. Eine Dynlink-Routine wird nur ausgeführt, da ein Thread (der zu einem Klientenprozeß gehört) den Dynlink-Code anspricht. Der Dynlink-Code wird als Kliententhread ausgeführt (aus der Perspektive des Systems betrachtet), die Dynlink-Routine ist lediglich eine vom Prozeß aufgerufene Unterroutine. Bevor der Klientenprozeß ein Dynlink-Paket aufrufen kann, stellt OS/2 sicher, daß die Segmente der Dynlink-Routine sich im adressierbaren Bereich des Klientenprozesses befinden. Keine Ring- oder Kontextumschaltung tritt beim Aufruf einer Dynlink-Routine auf. Ein *far*-Aufruf zum Einsprungspunkt einer Dynlink-Routine ist lediglich ein *far*-Aufruf einer Unterroutine im adressierbaren Bereich des Prozesses.

Ein Nebeneffekt besteht darin, daß Dynlink-Aufrufe sehr schnell sind, da wenig CPU-Zeit zum Zugriff auf das Dynlink-Paket benötigt wird. Ein weiterer Nebeneffekt ist, daß es keine Unterscheidung zwischen den Segmenten des Klientenprozesses und den Segmenten eines Dynlink-Prozesses gibt, da Segmente Prozessen zugehören und nur ein einziger Prozeß sowohl den Klienten- als auch den Dynlink-Code ausführt. Dasselbe gilt für Dateinummern, Semaphoren und ähnliches.

Hinweis: Bei Untersystem-Dynlink-Paketen kann die fehlende Trennung zwischen den Segmenten von Klient und Dynlink-Paket unter Umständen zu Problemen führen. Detaillierte Informationen finden Sie unter 7.11.1.

7.5 Daten

Bei den bisherigen Ausführungen zur dynamischen Verknüpfung wurde ein wichtiger Aspekt ausgelassen: Wie steht es mit der Behandlung der Daten der Dynlink-Routine? Für statisch verknüpfte Unterroutinen mit eigenen statischen Daten ergibt sich kein Problem. Der Linker verknüpft den externen Code mit dem Hauptcode und überprüft dabei den Datenbedarf des externen Codes. Der benötigte Bereich wird freigehalten. Referenzen des externen Codes auf seine Daten werden entsprechend gesetzt. Da der Linker alle .OBJ-Dateien zu einer .EXE-Datei verknüpft, können die statischen Segmente zwischen den einzelnen Elementen aufgeteilt werden.

Diese Methode ist bei der Verwendung von Dynlink-Routinen nicht einsetzbar, da deren Programmcode und Datenanforderungen beim Linken nicht bekannt sind. Es wäre möglich, die Dynlink-.OBJ-Datei zu erweitern, so daß der benötigte statische Datenbereich des Dynlink-Paketes dadurch beschrieben wird. Dies funktioniert aber nicht. Im übrigen: Selbst wenn es funktionieren würde, wäre es eine "armselige" Lösung, weil die Möglichkeit zur Aktualisierung des Dynlink-Codes dadurch eingeschränkt würde. Da der Hauptprogrammcode jeder Anwendung unterschiedliche Datenbereiche benötigt, würde der für das Dynlink-Paket reservierte Datenbereich bei jeder .EXE-Datei bei einem anderen Offset beginnen. Bei der Ausführung dieser .EXE-Dateien würden die Dynlink-Codesegmente Zugriff auf Daten verlangen, die für verschiedene Klientenprozesse in unterschiedlichen Segmenten vorliegen. Die Neuberechnung der statischen Referenzen in allen Dynlink-Modulen bei jedem Auftreten einer Kontextumschaltung wäre absurd.

Eine Alternative zur Zuweisung statischer Datenbereiche an Dynlink-Routinen bestünde darin, daß die aufrufenden Programme die entsprechenden Datenbereiche für die Dynlink-Routine belegen und bei jedem Aufruf Zeiger darauf übergeben. Dies Überlegung wurde jedoch bei der Planung von OS/2 rasch verworfen, da sich die *Call*-Anweisung für Dynlink-Routinen in diesem Falle von einer *Call*-Anweisung für normale

Unterroutinen unterscheiden würde. Hinzu kommt, daß die Untersysteme nicht unterstützt würden. Statt dessen macht OS/2 von der segmentierten Architektur des 80286 Gebrauch. Jede Dynlink-Routine kann ein oder mehrere Datensegmente zur Aufnahme ihrer statischen Daten verwenden. Jeder Klientenprozeß verfügt über einen eigenen Satz dieser Segmente. Da die Segmente nur Daten von Dynlink-Routinen enthalten dürfen (und keine Daten des aufrufenden Prozesses), ändern sich die Offsets der Daten im Segment beim Aufruf durch unterschiedliche Klientenprozesse nicht. Es muß nur sichergestellt sein, daß die Segmentselektoren der statischen Datensegmente der Dynlink-Routine für jeden Klientenprozeß identisch sind.

OS/2 sichert dies durch die Methode der reservierten LDT-Einträge. An dieser Stelle soll keine allgemeine Einführung in die segmentierte Architektur des 80286 gegeben werden, eine kurze Zusammenfassung scheint aber sinnvoll. Jeder Prozeß im Protect-Modus des 80286 besitzt ein Maximum von 16.383 Segmenten. Die Segmente werden durch zwei Tabellen beschrieben: die LDT (*Local Descriptor Table*) und die GDT (*Global Descriptor Table*). Eine Anwendung kann diese Tabellen weder lesen noch darin schreiben; sie werden von OS/2 verwaltet. Der Mikroprozessor 80286 nutzt die Inhalte beim Laden von Selektoren durch die Prozesse in deren Segmentregister.

In der Praxis wird die GDT nicht für Anwendungsprozesse verwendet, es bleiben die 8192 Segmente der LDT - genauer gesagt, die 8192 Segmentselektoren, die OS/2 verwenden kann, um auf Speichersegmente zu zeigen. Der 80286 unterstützt positionsunabhängigen Code nicht effizient, so daß 80286-Programme als Teil des Befehlsstroms auch Segmentselektoren enthalten, die zum Zugriff auf Speicherstellen benötigt werden (gleiches trifft für den Offset in einem Segment zu). Dies gilt sowohl für Code- als auch für Datenreferenzen.

Wenn OS/2 ein Programm in den Speicher lädt, beschreibt die .EXE-Datei die Anzahl, den Typ und die Größe der Programmsegmente. OS/2 reserviert die Segmente und belegt für jedes Segment einen der 8192 möglichen LDT-Selektoren. Hier

gibt es keine Konflikte zwischen den Prozessen in einem System, da jeder Prozeß über eine eigene LDT verfügt und daher einen eigenen Satz von 8192 LDT-Selektoren besitzt. Nachdem OS/2 für jedes Segment (sowohl Code- als auch Datensegmente) einen Selektor ausgewählt hat, wird eine in der .EXE-Datei bereitgestellte Adreßtabelle zur Neuberechnung jeder Segmentreferenz verwendet. Dabei werden die durch den Linker eingefügten Platzhalterwerte auf den Wert des jeweilgen Segmentselektors verändert. OS/2 verknüpft oder zerlegt Segmente unter keinen Umständen, so daß der Offset-Teil der Adressen niemals verändert wird (nur der segmentierte Teil). In einem Programm treten mehr Adreßoffsets als Segmentreferenzen auf. Aus diesem Grund benötigt der Neuberechungsvorgang unter OS/2 nicht viel Zeit.

Stellt OS/2 fest, daß der zu ladende Prozeß Referenzen auf eine Dynlink-Routine besitzt, wird die Situation komplizierter. Wird dabei eine Dynlink-Routine angesprochen, die bereits von einem anderen Prozeß geladen wurde, so befindet sich diese im Speicherbereich und hat bereits bestimmte LDT-Einträge der LDT des vorigen Klientenprozesses belegt. OS/2 muß dieselben Einträge in der LDT des neuen Prozesses mit Zeigern auf die Dynlink-Routine füllen. Es ist nicht möglich, andere LDT-Einträge und deren Daten für die Routine zu verwenden, da deren Segmentadressen bereits durch den früheren, aufrufenden Prozeß festgelegt sind. Ein Problem tritt auf, wenn die benötigten Einträge bereits durch andere Routinen belegt sind (das gilt sowohl für Daten- als auch für Codesegmente der Dynlink-Routine).

Zur Lösung dieses Problems werden die reservierten LDT-Einträge benötigt. OS/2 reserviert viele der LDT-Einträge jedes Prozesses als *Disjoint*-Bereich (Raum für gemeinsam genutzte Objekte). Die gleichen Eintragsnummern werden in der LDT jeden Prozesses für eine bestimmte Dynlink-Routine bereitgestellt. Bei der Belegung eines LDT-Selektors für ein Speichersegment, das von mehreren Prozessen angesprochen werden kann, wird der Eintrag in einem reservierten LDT-Eintrag erfolgen. Nach der Belegung des Selektors werden alle Einträge in

allen anderen LDTs des Systems für diese Routine reserviert. Falls die zugehörige Dynlink-Routine von einem Prozeß nicht verwendet wird, bleibt dieser Eintrag in dessen LDT leer (das heißt ungültig), sie wird für keinen anderen Zweck eingesetzt. Wird die entsprechende Dynlink-Routine benötigt, zeigt der gleiche Eintrag der LDT anderer Prozesse auf dasselbe Segment, das durch den ersten aufrufenden Prozeß festgelegt wurde. Dadurch ist sichergestellt, daß auch nach Stunden noch alle gemeinsam zu verwendenden Segmente von jedem Prozeß aus angesprochen werden können (mit einem *DosLoadModule*-Aufruf). OS/2 findet die entsprechenden Einträge in der LDT des Prozesses, die nur auf diese Aufgabe warten. Der reservierte LDT-Bereich wird für alle gemeinsam genutzten Speicherobjekte verwendet, nicht nur für Dynlink-Routinen. Gemeinsam verwendete Datensegmente werden ebenfalls im reservierten LDT-Bereich belegt. Die Codesegmente eines Prozesses werden dort nicht belegt, können aber dennoch gemeinsam verwendet werden. In Bild 7.4 wird das Konzept der reservierten LDT-Bereiche verdeutlicht. Die Punkte in den schattiert dargestellten Selektoren zeigen reservierte, aber bislang noch ungültige Selektoren an. Sie sind für den Fall reserviert, daß der Prozeß später den Zugriff auf ein gemeinsam verwendetes Segment verlangt (dem diese reservierten Einträge zugeordnet sind). Nur Prozeß A verwendet das Dynlink-Paket DLX, so daß die entsprechenden reservierten LDT-Einträge in der LDT des Prozesses B gleichfalls reserviert sind. Beide Prozesse verwenden das Dynlink-Paket DLY.

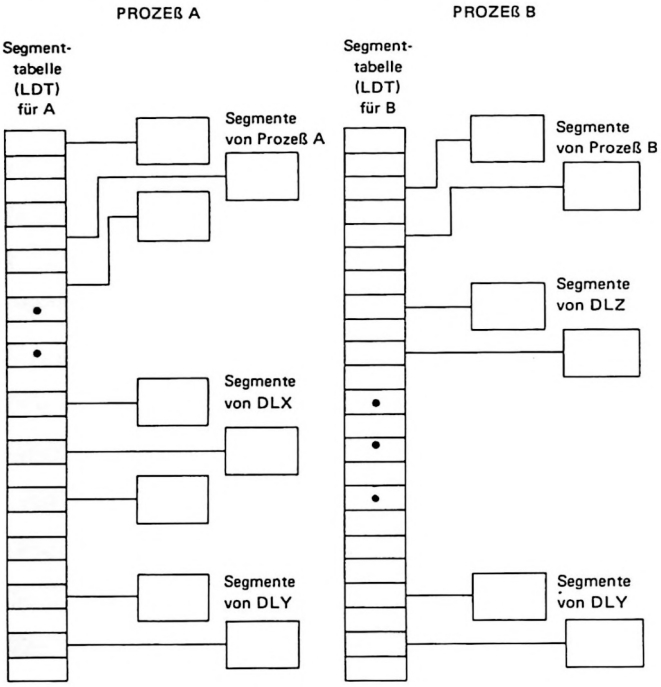


Bild 7.4 Der Disjoint-LDT-Bereich

7.5.1 Instanzdaten

OS/2 unterstützt zwei Arten von Datensegmenten für Dynlink-Routinen: Instanz- und globale Daten. In Instanzdatensegmenten stehen für jede Instanz der Dynlink-Routine spezifische Daten. Mit anderen Worten: Eine Dynlink-Routine verfügt für jeden Prozeß, der sie nutzt, über einen eigenen Satz von Instanzdatensegmenten. Der Dynlink-Code kann die Daten problemlos adressieren, die Segmentselektoren lassen sich als direkte Werte

ansprechen. Linker und Lader von OS/2 sorgen dafür, daß die Selektoren bei der Ausführung an den richtigen Stellen stehen.

Die Verwendung der Instanzdaten ist sowohl für den Klientenprozeß als auch für den Dynlink-Code praktisch unsichtbar. Der Klientenprozeß ruft die Dynlink-Routine auf, das/die Instanzdatensegment(e) der Routine ist/sind dabei völlig unerheblich. Die Dynlink-Routine kann Adressen in ihrem Instanzdatenbereich an den Klientenprozeß weitergeben. Der Klient kann nicht zwischen statischen und dynamischen Dynlink-Routinen unterscheiden. Auch im Code der Dynlink-Routine sind keine besonderen Vorkehrungen für die Instanzdatenregister notwendig. Linker und OS/2 sorgen für die Belegung der LDT-Selektoren für gemeinsam genutzte Objekte, laden die Segmente, stellen die Referenzen her usw.

Eine Dynlink-Routine, die nur Instanzdatensegmente (oder gar keine Datensegmente) verwendet, läßt sich als ein einziges Klientenpaket schreiben, also genau wie eine statisch verknüpfte Unterroutine. Eine solche Dynlink-Routine mag mehrere Klientenprozesse haben, für die Routine ist dies jedoch nicht ersichtlich. Jeder Prozeß verfügt über einen eigenen Satz an Instanzdatensegmenten. Zum Anlegen eines neuen Satzes lädt OS/2 die Originalsegmente aus der .DDL-Datei. Die Streuung der Codesegmente durch OS/2 hat keine Auswirkungen auf die Funktionalität der Routine selbst.

7.5.2 Globale Daten

Neben den Instanzdatensegmenten stehen den Dynlink-Routinen Globaldatensegmente zur Verfügung. Ein Globaldatensegment ist nicht einer bestimmten Instanz zugeordnet, sondern für alle Instanzen gleichermaßen verfügbar. Für jede Dynlink-Routine existiert daher maximal ein Satz an Globaldatensegmenten, auf die alle Instanzen der Routine zugreifen. Das Segment wird nur einmal beim ersten Linken der Dynlink-Routine mit einem Prozeß geladen und steht von diesem Zeitpunkt an im Speicher bereit. Globaldatensegmente schaffen sichbare Verbindungen zwischen den Klientenprozessen und der Dynlink-Routine. Sie die-

nen in erster Linie der Unterstützung von Subsystemen (dies wird noch näher erläutert). Bild 7.5 zeigt eine Dynlink-Routine mit Instanz- und Globaldatensegmenten.

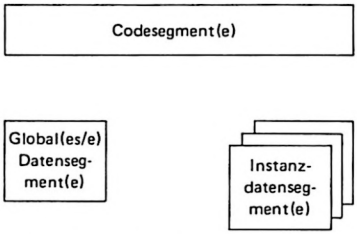


Bild 7.5 Dynlink-Segmente

7.6 Dynlink-Pakete als Unterroutinen

Dynlink-Routinen (oder Pakete) fallen grundsätzlich in eine von zwei Kategorien: Unterroutinen oder Untersysteme (*subroutines* oder *subsystems*). Wie dargelegt wird eine dynamische Verknüpfungsroutine in ähnlicher Weise verfaßt und ausgeführt wie eine statische Verknüpfungsroutine. Die einzigen Unterschiede bestehen in der Vorbereitung der Dynlink-Bibliotheksdatei, die die Routinen enthält, und in der Vorbereitung der speziellen .OBJ-Datei, zu der Klientenprozesse gebunden werden. Bei der Ausführung können Dynlink- und Klientenroutinen ihre eigenen statischen Daten unabhängig voneinander verwenden. Zeiger auf die Datenbereiche lassen sich wechselseitig übergeben. In diesem Modell existiert nur ein gravierender Unterschied zwischen statischer und dynamischer Verknüpfung: Die Dynlink-Routine kann keinen Bezug auf externe Symbole des Klientencodes nehmen. Umgekehrt gibt es für den Klientenprozeß keine Möglichkeit, externe Referenzen auf Dynlink-Symbole aufzubauen (abgesehen vom Dynlink-Einsprungspunkt). Die Ausführungsumgebung entspricht fast völlig der einer herkömmlichen statisch gebundenen Unterroutine: Klient und Unterroutine haben jeweils ihre eigenen statischen Datenbereiche, die beide zum Adreßbereich des Prozesses gehören. Dynlink-

Routinen und Applikationen können Daten miteinander austauschen, allerdings nur durch die Übergabe von Zeigern auf die eigenen Daten.

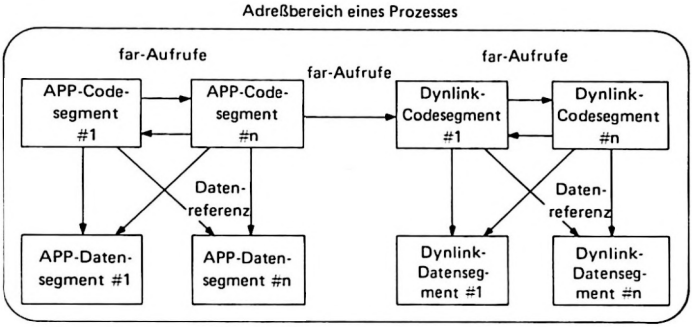


Bild 7.6 Dynlink-Routinen als Unterroutinen

7.7 Unterroutinen

Der Ausdruck *Dynlink-Untersystem* (*dynlink subsystem*) bezieht sich auf die geplante Verwendung einer bestimmten Art von Dynlink-Paketen. OS/2 trifft keine Unterscheidung zwischen Untersystemen und Unterroutinen.

Unter einem Untersystem versteht man ein Dynlink-Modul, das eine Reihe von Funktionen auf der Basis gemeinsamer Ressourcen enthält. Ein Beispiel stellen die VIO-Routinen von OS/2 für die Bildschirmsteuerung dar. Ein Untersystem umfaßt im allgemeinen einen Satz zusammengehöriger Funktionen, die einer beliebigen Anzahl von Klienten zur Verfügung gestellt werden. Das VIO-Paket enthält einen Bildschirm-Controller mit einer kleinen Anzahl von Bildschirmgruppen für beliebig viele Klientenprozesse.

Da die verschiedenen Routinen eines Untersystems logisch zusammengehören, müssen sie ihre Daten gegenseitig austauschen können. Man nimmt hierfür Globaldatensegmente, in denen Informationen über die kontrollierten Ressourcen (etwa den Bildschirm) abgelegt werden. Untersysteme verfügen zudem über

Puffer, Flaggen, Semaphoren und ähnliches. Für die einzelnen Klienten werden Instanzdatensegmente angelegt, von der Benutzung von Globaldatensegmenten ist zu diesem Zweck dringend abzuraten. Ein Dynlink-Untersystem unterscheidet sich also von einer Dynlink-Unterroutine lediglich durch ein zusätzliches statisches Datensegment.

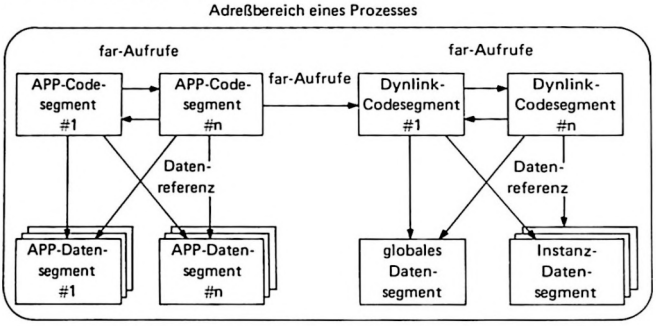


Bild 7.7 Dynlink-Routinen als Untersysteme

7.7.1 Unterstützung für Untersysteme

Es gibt zwei OS/2-Leistungsmerkmale, die für den Einsatz von Untersystemen besonders nützlich sind: Globaldatensegmente (wurde bereits erläutert) und spezielle Unterstützung zur Initialisierung und Beendigung von Klientenprozessen. Wenn ein Untersystem seine Klienten in einem Globaldatensegment verwaltet, muß es natürlich über das Hinzukommen und Wegfallen von Klienten Bescheid wissen. Das einfache Dynlink-Routinenmodell enthält hierfür keine adäquaten Kommunikationsmöglichkeiten. Ein Untersystem enthält zwar auch in diesem Modell Einsprungspunkte für die Initialisierung und Beendigung von Prozessen, aber was hindert einen Prozeß an der Beendigung ohne Aufruf des entsprechenden Dynlink-Einsprungspunktes. Freilich wäre dies ein Fehler des Klienten; aber das Betriebssystem müßte diesen Fehler entdecken und beheben können. Es ist akzeptabel, daß die Funktionalität des Dynlink-Untersystems (und damit die Funktionalität aller angeschlossenen Klienten) gefährdet ist, wenn in einem Klientenprozeß ein Fehler auftritt.

Es gibt zwei Arten der Initialisierung von Untersystemen: global und als Instanz. Ein Untersystem kann auf die eine oder auf die andere Weise starten, aber nicht beide Möglichkeiten zu einem Zeitpunkt nutzen. Bei globaler Initialisierung wird der Einsprungspunkt nur einmal bei jeder Systemaktivierung angesprochen. Bei der ersten Bezugnahme auf das Dynlink-Untersystem belegt OS/2 das oder die Globaldatensegmente des Pakets mit den Originalwerten aus der .DLL-Datei. Anschließend ruft OS/2 das Routinenpaket am entsprechenden Einsprungspunkt auf. Der hierfür erforderliche Thread gehört zum ersten Klientenprozeß (obgleich der Aufruf durch OS/2 implizit erfolgt), so daß zugleich das Instanzdatensegment für den ersten Klient eingerichtet wird. Dieses erste Instanzdatensegment läßt sich bereits für die globale Initialisierung verwenden. Dies bedeutet: Obgleich das Dynlink-Untersystem Dateien öffnen, lesen, schreiben und schließen kann, ist es möglicherweise nicht in der Lage, eine Dateinummer zu öffnen oder sie im Globaldatensegment für spätere Zwecke zu speichern.

Zur Erinnerung: Untersysteme verfügen nicht über Ressourcen, nur Prozesse besitzen Ressourcen! Wenn ein Dynlink-Paket eine Datei öffnet, so ist diese Datei nur für einen einzigen Klientenprozeß offen. Die Dateinummer ist nur gültig, sofern dieser Klient das Untersystem aufruft. Speichert ein Dynlink-Paket die Dateinummer von Prozeß A in einem Globaldatensegment und versucht anschließend bei der Bearbeitung von Prozeß B einen Lesevorgang mit dieser Dateinummer, kommt es zu einem Fehler.

Neben der Global- steht die Instanzinitialisierung zur Verfügung. Der Instanzeinsprungspunkt wird in gleicher Weise wie der Globaleinsprungspunkt aufgerufen, mit dem einzigen Unterschied, daß der Aufruf für jeden neuen Klienten zu Anfang erneut erfolgen muß. Die bereits existierenden Instanzdatensegmente werden verwendet. Sie erhalten ihre Anfangswerte aus der .DLL-Datei. Der Einsprungspunkt für eine ladedynamische Dynlink-Routine muß vor der Abarbeitung des Klientenprozesses aufgerufen werden. Der Einsprung in eine laufzeitdynamische Dynlink-Routine erfolgt beim Aufruf der *DosLoadModu-*

le-Funktion durch den Klienten. Die gleichzeitige Verwendung von Global- und Instanzinitialisierung ist nicht ratsam. Wird beides benötigt, sollte die Instanzinitialisierung spezifiziert und zugleich ein Zähler in einem der Globaldatensegmente eingerichtet werden, der die erste Instanzinitialisierung entdeckt.

Wichtiger als die Initialisierungskontrolle ist die Beendigungskontrolle. Ein Untersystem mag in seinen Globaldatensegmenten Datensätze, Puffer oder Semaphoren für einen Klientenprozeß haben. Möglicherweise stehen mehrere Anfragen eines Klienten in einer Warteschlange, die bei der Beendigung des Klienten gelöscht werden müssen. Ein Dynlink-Paket muß keine Freigabe der Instanzdatensegmente ausführen, da diese zum Klientenprozeß gehören. Bei der Beendigung des Klienten werden sie entfernt. Die Globaldatensegmente werden bei der Beendigung des letzten Klienten eines Dynlink-Moduls freigegeben. Möglicherweise sollen zuvor noch Aufgaben durchgeführt werden, etwa die Aktualisierung einer Journaldatei oder die Freigabe einer Systemsemaphore.

Eine Dynlink-Routine könnte die Funktion *DosSetSigHandler* verwenden, um die Beendigung eines Klientenprozesses zu erfahren. Dies sollte jedoch auf keinen Fall geschehen, da nicht in jedem Fall der Prozeßbeendigung ein entsprechendes Signal ausgegeben wird. Bei der Prozeßbeendigung durch den *DosExit*-Aufruf wird beispielsweise kein Beendigungssignal ausgegeben und die Dynlink-Routine somit nicht über das Prozeßende informiert. Hinzu kommt, daß pro Signalart und Prozeß nur ein Signal existiert. Da sich Dynlink-Routine und Klientenprozeß üblicherweise als "Black Box" betrachten (und auch betrachten sollten), kann es leicht zu Konflikten kommen, wenn beide auf dieselben Signale zugreifen. Dieser Konflikt kann auch zwischen zwei Dynlink-Routinen auftreten.

Es empfiehlt sich daher die Verwendung von *DosExitList*. Die Funktion erlaubt einem Prozeß die Spezifizierung von einer oder mehreren Unterroutinenadressen, die bei Beeendigung des Prozesses aufgerufen werden. *DosExitList* ist das ideale Werkzeug für die Prozeßbeendigungskontrolle. Die Adressenliste

kann sehr lang sein, und sie wird unter allen Umständen der Prozeßbeendigung abgearbeitet. Der Klientenprozeß und die von ihm aufgerufenen Dynlink-Untersysteme dürfen jeweils eigene Beendigungsroutinen haben. Weitere Erläuterungen zu *DosExitList* finden Sie unter 16.2.

7.8 Dynlinks als Prozeßschnittstellen

Wie bereits ausgeführt, haben Dynlink-Routinen Probleme mit der Handhabung von Ressourcen (abgesehen vom gemeinsamen Hauptspeicher), da Ressourcen in OS/2 prozeßbezogen vergeben werden. Welche Dateien für eine Dynlink-Routine geöffnet sind, welche Semaphoren ihr gehören und anderes hängen davon ab, welcher Prozeß gerade aktiv ist. Selbst die Verwaltung des gemeinsamen Datenspeichers ist schwierig, da alle Klienten darauf Zugriff nehmen können. Der Speicher bleibt erhalten, bis der letzte Klientenprozeß der Dynlink-Routine beendet ist.

Ein Weg, diese Probleme in den Griff zu bekommen, besteht darin, die Dynlink-Routine als Frontend für einen Serverprozeß einzusetzen. Bei der Initialisierung des Moduls prüft dieses anhand einer Systemsemaphore, ob der Serverprozeß schon aktiv ist und startet ihn gegebenenfalls. Dies muß mit der "abgekoppelten" Form von *DosExecPgm* erfolgen, damit der Serverprozeß nicht dem ersten Klienten des Untersystems als Kindprozeß zugeordnet wird. Ein derartiger Fehler könnte dazu führen, daß der Vaterprozeß niemals abbricht, da der Serverprozeß als Teil des Befehlsbaums erscheint (siehe Bild 7.8).

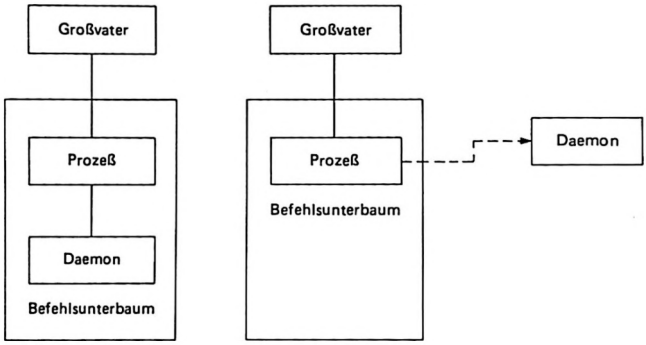


Bild 7.8 Dynlink-Deamon-Initiierung

Läuft der Serverprozeß, kann das Dynlink-Untersystem einige oder alle Anfragen an ihn über die zahlreichen IPC-Einrichtungen von OS/2 richten. Beispiel: Für ein Datenbanksystem bietet sich der Einsatz eines dedizierten Serverprozesses an, um die Datenbank offenzuhalten und Schreib-/Leseoperationen auszuführen. Puffer und ISAM-Verzeichnisse wird man in einem gemeinsamen Speichersegment halten, auf das das Dynlink-Untersystem für jeden Klienten zugreift. Anfragen nach Daten, die sich im Puffer befinden, benötigen keinen IPC-Zugriff auf den Serverprozeß.

Es gibt Dynlink-Pakete, die nur als prozedurale Schnittstelle zwischen Prozessen fungieren. Auch hierzu ein Beispiel: Ein Tabellenkalkulationsprogramm soll mit einer Schnittstelle ausgestattet werden, von der andere Programme die aktuellen Daten aus der Kalkulationstabelle übernehmen können. Die beste Lösung: Dem Tabellenkalkulationsprogramm wird eine Dynlink-Bibliothek mitgegeben, die Klientenprozessen eine prozedurale Schnittstelle zum Tabellenkalkulationsprozeß zur Verfügung stellt. Die Bibliotheksroutine ruft eine nicht aktive Kopie der Kalkulationstabelle auf (vielleicht ein Auszug aus der .EXE-

Datei), um die Informationen zu erhalten und per IPC an den jeweiligen Klientenprozeß weitergeben zu können.

Eine andere Möglichkeit besteht darin, den Zugriffscode, der das Datenformat der Kalkulationstabelle versteht, direkt in das Dynlink-Paket aufzunehmen. Bei der Auslieferung einer neuen Version des Tabellenkalkulationsprogramms kann einfach eine neue Version des Dynlink-Paketes mitgegeben werden. In diesem Fall könnte das Tabellenkalkulationsprogramm die Dynlink-Routinen unmittelbar verwenden, und der Umweg über eine Tabellenkopie könnte entfallen. Die Details der Implementierung bleiben dem Klientenprozeß in jedem Fall verborgen, er führt lediglich einen Aufruf durch, der die gewünschten Daten liefert.

Aus der Vogelperspektive betrachtet ist dies eine einfache Regelung: Ein Klientenprozeß erhält durch eine Unterroutinenbibliothek über IPC-Services von einem Serverprozeß eine Dienstleistung. Aus der näheren Sicht des Programmierers ist die Schnittstelle zum Dynlink-Untersystem von entscheidender Bedeutung. So läßt sich beispielsweise ein Serverprozeß in einer neuen Version leistungsfähiger gestalten und mit anderen IPC-Systemen ausrüsten, ohne daß die Binärkompatibilität mit der bestehenden Klientenbasis verlorengeht. In Bild 7.9 ist ein Dynlink-Paket zu sehen, das als Schnittstelle zu einem Daemon-Prozeß fungiert. Der Datenaustausch erfolgt über gemeinsam genutzten Speicher und andere Möglichkeiten.

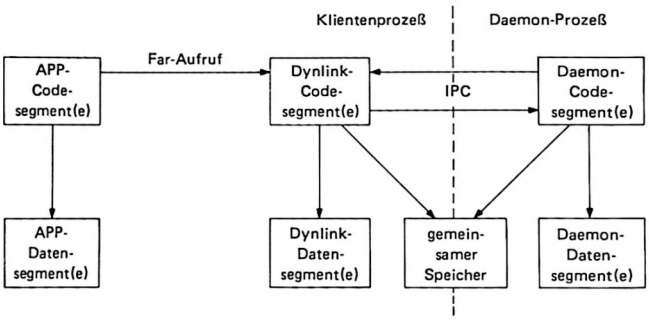


Bild 7.9 Dynlink-Routinen als Daemon-Prozeß-Schnittstellen

7.9 Dynlinks als Schnittstellen zum Kernel

Auf den vorangehenden Seiten wurde die Verwendung von Dynlink-Bibliotheken als Unterroutinenbibliotheken, Untersystemen und Schnittstellen zwischen Prozessen erläutert. Dynlink-Bibliotheken lassen sich darüber hinaus auch als Schnittstellen zu OS/2 selbst einsetzen.

Eine Reihe von OS/2-Aufrufen sind als einfache Bibliotheksroutinen implementiert. Ein Beispiel hierfür stellt *DosErrClass* in OS/2 Version 1.0 dar. Die Funktion übernimmt einen Fehlercode und lokalisiert in einer Tabelle einen zugehörigen Textstring, eine Klassifizierung des Fehlers und einen Hinweis auf die empfohlene Aktion. Derartige Services gehören traditionell zum Kernel des Betriebssystems, und zwar nicht etwa, weil dies programmtechnisch sinnvoll ist, sondern weil sich die Fehlertabellen mit jeder neuen Betriebssystemversion ändern. Stünden die Services als statisch verknüpfte Unterroutinen zur Verfügung, erhielten ältere Applikationen unter neueren Betriebssystemversionen neue Fehlercodes, die in der Codetabelle der Bibliothek fehlten.

Obgleich *DosErrClass* in OS/2 als Bibliotheksroutine implementiert ist, gibt es das genannte Problem nicht, da es sich um eine Dynlink-Bibliotheksroutine handelt und die .DLL-Datei zum Betriebssystem gehört. Künftige Versionen des Betriebssystems werden erweiterte Versionen von *DosErrClass* enthalten, die die neuen Fehlercodes kennen. Das Konzept der dynamischen Verknüpfungen gibt OS/2 somit ein hohes Maß an Flexibilität bezüglich der Bündelung von Funktionalität.

Funktionen wie *Datei öffnen* oder *Speicher belegen* kann man nicht als einfache Unterroutinen implementieren. Diese Funktionen benötigen Zugang zu wichtigen internen Datenstrukturen, die selbstverständlich geschützt sind, so daß sie nur durch privilegierten Code geändert werden können. Zur Erreichung dieser Services muß der Prozessor einen Systemaufruf durchführen, den Kernel in kontrollierter Weise ansprechen und dort mit genügender Privilegierung ablaufen. Der Privilegübergang erfolgt über ein *Call Gate*, das ein Leistungsmerkmal der Prozessoren 80286 und 80386 darstellt. Ein Programm spricht den *Call-Gate*-Aufruf wie jeden anderen *far*-Aufruf an, spezielle Flaggen in der GDT und LDT informieren den Prozessor darüber, daß es sich um einen Gate- statt um einen normalen Aufruf handelt.

In OS/2 sind Systemaufrufe nicht von Dynlink-Aufrufen zu unterscheiden. Alle OS/2-Systemaufrufe sind im Dynlink-Modul *DosCalls* definiert. Bei der Bezugnahme auf dieses Modul durchsucht OS/2 eine in das Betriebssystem eingebaute Tabelle nach residenten Funktionen. Ist eine Funktion nicht in der Liste, wird eine normale Dynlink-Verknüpfung durchgeführt. Steht die Funktion hingegen in der Liste, initiiert OS/2 einen Gate-Aufruf. Die Durchlässigkeit zwischen Bibliothek und Call-Gate-Funktionen erklärt im übrigen auch, warum die Übergabe einer ungültigen Adresse in einem OS/2-Systemaufruf für den aufrufenden Prozeß zu einem GP-Fehler führt. Da der OS/2-Kernel-Code den GP-Fehlermechanismus kontrolliert, könnten OS/2-Aufrufe über das Gate leicht einen Fehlercode liefern, wenn eine ungültige Adresse zu einem GP-Fehler führt. Wäre dies der Fall, würde das Verhalten von OS/2-Aufrufen

von der Art der Implementierung abhängen: Dynlink-Aufrufe würden bei ungültigen Adressen zu GP-Fehlern führen (die Anweisungen LAR und LSL würden als Abhilfe nicht ausreichen, da ein anderer Prozeßthread ein Segment nach einer LAR-Instruktion, aber noch vor der Referenz freigeben könnte), Call-Gate-Aufrufe würden einen Fehlercode liefern. In OS/2 wurde diese Unterscheidung bewußt vermieden, um damit auch die Freiheit zu lassen, in künftigen Versionen Funktionen zwischen Call-Gate- und Dynlink-Implementierung auszutauschen. Da Dynlink-Aufrufe, die nicht über das Call Gate laufen, bei ungültigen Adressen auf jeden Fall einen GP-Fehler verursachen müssen, wurde dies auch für Call-Gate-Aufrufe vorgesehen.

7.10 Dynlinks als Teil der Betriebssystemarchitektur

Dynamische Verknüpfungen spielen in OS/2 in dreierlei Hinsicht eine wichtige Rolle: als Systemschnittstelle, als Einheitenschnittstelle mit großer Bandbreite und als Hilfsmittel zur Anbindung von Servicepaketen außerhalb des Systemkernels.

Die Rolle von Dynlinks als Systemschnittstelle ist klar. Sie bieten hier eine einheitliche, hocheffiziente Schnittstelle sowohl zum Systemkernel als auch zu Funktionen außerhalb des Kernels. Die Schnittstelle ist unmittelbar kompatibel mit höheren Programmiersprachen und nutzt die Besonderheiten der Prozessoren 80286 und 80386 (insbesondere die automatische Argumentübergabe bei Aufrufen an Ring-0-Kernel-Code). Das Dynlink-Konzept stellt ein komfortables Konzept der Namensgebung zur Verfügung. Es erlaubt weiterhin die Verteilung von Funktionen zwischen Bibliothek und Kernel. Und schließlich ist das Konzept außerordentlich erweiterungsfähig.

Dynlinks dienen jedoch nicht nur als Systemaufrufe. In den vorderen Kapiteln des vorliegenden Buches habe ich die Forderung nach Einheitenschnittstellen erhoben, die so einheitenunabhängig wie Schnittstellentreiber sind, jedoch nicht viel "Wasserkopffunktionalität" enthalten. Die dynamische Verknüpfung

stellt genau diese Schnittstelle bereit. Dynlinks ermöglichen Anwendungsprogrammen den schnellen Zugang zu Routinenpaketen, die unmittelbar auf Einheiten zugreifen können (Kapitel 18). Der Aufruf ist schnell und in weitem Maße parametrisierbar. Zudem können Dynlink-Aufrufe ohne Privileg- oder Ring-Umschaltung erfolgen. Das Dynlink-Paket kann direkt auf die Datenbereiche des Klienten zugreifen. Schließlich vermögen Dynlinks über Untersystemleistungsmerkmale Einheiten mehreren Klienten virtuell zur Verfügung zu stellen. Die Einheitenunabhängigkeit ergibt sich aus der Tatsache, daß man zu jeder neuen Einheit ein neues Dynlink-Paket installieren kann. Einen gravierenden Nachteil hat es allerdings, wenn man Einheitentreiber durch Dynlink-Treiber ersetzt: Dynlinks vermögen keine Hardwareinterrupts zu erkennen. Einheiten wie etwa Bildschirme erzeugen keine Interrupts. Bei interruptgesteuerten Einheiten wird jedoch weiterhin ein echter Einheitentreiber benötigt. Dieser Treiber kann sämtliche Einheitenschnittstellenfunktionen enthalten, die Funktionalität läßt sich aber auch zwischen Einheitentreiber und Dynlink-Paket aufteilen. In diesem Fall fungiert das Dynlink-Paket als Frontend zum Treiber (Kapitel 17 und 18).

Dynlink-Routinen lassen sich auch zur Bereitstellung von Servicepaketen außerhalb des Kernel heranziehen - als eine offene Systemarchitektur für Software. Die meisten Betriebssysteme entsprechen in einem Punkt den ersten Versionen des Systems im Apple Macintosh: Es sind geschlossene Systeme. Betriebssystemerweiterungen können nur durch die Entwickler vorgenommen werden. OS/2 ist hingegen ein offenes Betriebssystem. Software- und Systemhäuser, aber auch Anwender, können Systemservices in Form von Dynlink-Modulen nach Belieben hinzufügen, ähnlich wie in einen offenen PC Hardwarekarten eingesteckt werden können. Diese Analogie läßt sich noch fortsetzen. Einige Hardwarekarten fanden derart weite Verbreitung, daß ihre Schnittstellen heute als Standard gelten. Beispiele hierfür stellen die Modems der Firma Hayes und die Graphikkarten von Hercules dar. Es ist anzunehmen, daß um OS/2 herum ebenfalls neue Standardschnittstellen durch populäre Dynlink-

Pakete entstehen. So ist davon auszugehen, daß einige Anbieter in der Zukunft mit Datenbank-Dynlink-Routinen auf den Markt kommen werden, die der Standarddatenbank-Dynlink-Schnitt +stelle entsprechen, jedoch besser, billiger und schneller sind.

Selbstverständlich erlaubt das Dynlink-Konzept auch den OS/2-Entwicklern selbst das Hinzufügen neuer Systemmerkmale. Die Dynlink-Schnittstelle ermöglicht die Implementierung neuer Funktionalität als Unterroutinen oder Prozesse, sogar über eine Netzwerkumgebung verteilt.

7.11 Details der Implementierung

Obgleich Dynlink-Routinen häufig wie herkömmliche statische Unterroutinen eingesetzt werden, gibt es doch eine Reihe von Punkten, die der Programmierer beachten sollte. Im folgenden werden die Aspekte aufgeführt, die ein gutes von einem schlechten Dynlink-Paket unterscheiden.

7.11.1 Dynlink-Datensicherheit

Es wurde bereits erläutert, wie ein Dynlink-Paket als Unterroutine eines Klientenprozesses abläuft und wie der Prozeß Zugang zu den Instanz- und Globaldatensegmenten der Dynlink-Routinen findet. Zur Wiederholung: Der Klientenprozeß hat adressierbaren Speicherzugriff auf alle Globaldatensegmente des Dynlink-Paketes, aber nur auf die Instanzdatensegmente, die ihm zugeordnet sind. Diese Methode ist sehr effizient und damit vorteilhaft, sie hat jedoch einen Nachteil, der unter Umständen schwer wiegen kann: Die einzelnen Klientenprozesse können das Globaldatensegment durcheinanderbringen.

Im allgemeinen ist die Gefahr für die Beeinträchtigung des Globaldatensegments gering. Solange die Dynlink-Routine dem Klientenprozeß nicht die entsprechenden Zeiger übergibt, kann der Klient nicht auf die Datensegmentselektoren des Pakets zugreifen. Ein Prozeß, der die Zeiger nicht kennt, könnte höchstens durch Zufall einen Wert finden, der zum Selektor eines

Dynlink-Paketes führt. Da die meisten Selektorwerte ungültig sind, ist die Wahrscheinlichkeit recht gering, daß durch Zufall ein Datensegmentselektor angesprochen wird, bevor ein Codesegment- oder ein ungültiger Selektor erzeugt wird. Bei einem Codesegment- oder einem ungültigen Selektor wird der Prozeß aufgrund eines GP-Fehlers sofort beendet.

Die Anforderungen an die Datensicherheit hängen unmittelbar mit der Art des Dynlink-Paketes zusammen. Bei einem Dynlink-Paket ohne Globaldatensegment gibt es überhaupt kein Risiko in dieser Hinsicht; allerdings kann ein Klientenprozeß die Daten im Instanzdatensegment zerstören, dies ist jedoch die natürliche Folge eines Programmfehlers. In einem Dynlink-Paket mit Globaldatensegment läßt sich das Datenrisiko mindern, indem das Paket keine Zeiger auf das Segment an die Klientenprozesse weiterreicht. Ist das Segment nicht zu umfangreich, läßt sich die Datensicherheit im Globaldatensegment auch mit einer Prüfsumme erhöhen, sofern es wichtig ist, einen Datenfehler rasch zu entdecken.

Bei besonders hohen Ansprüchen bezüglich der Datensicherheit kann man ein Dynlink-Paket in Verbindung mit einem speziellen dedizierten Prozeß einsetzen, wie oben beschrieben. Die sensitiven Daten werden ausschließlich im dedizierten (das heißt separaten) Prozeß gehalten und dem Dynlink-Paket auf IPC-Basis klientenweise zur Verfügung gestellt. Die Segmente des dedizierten Prozesses sind somit vor dem Zugriff durch die Klientenprozesse des Dynlink-Paketes unter allen Umständen geschützt.

7.11.2 Dynlink: Nutzung gemeinsamer Ressourcen

Bislang war von dem gemeinsamen Zugriff auf *Pure*-Segmente die Rede. Die Möglichkeit zur gemeinsamen Nutzung von *Pure*-Segmenten stellt eine Optimierung dar, die OS/2 für alle Speichersegmente bietet, gleichgültig, ob es Dynlink- oder Anwendungssegmente sind. Unter einem Pure-Segment versteht man ein Segment, das in seiner Lebensspanne zu keinem Zeitpunkt verändert wird. Alle Codesegmente (mit Ausnahme der durch

DosCreateCSAlias erzeugten) sind Pure-Segmente. Nur-lese-Datensegmente sind ebenfalls Pure-Segmente. Wenn OS/2 zwei Kopien des gleichen Pure-Segments lädt, führt das Betriebssystem "hinter den Kulissen" einen Optimierungsvorgang durch und gibt dem zweiten Klienten Zugang auf die erste Kopie des Segments, so daß keine Speicherkapazität für eine zweite Kopie verschwendet wird.

Ein Beispiel: Wenn zwei Kopien eines Programms ablaufen, so daß alle Codesegmente Pure-Segmente sind, wird OS/2 in den meisten Fällen nur eine Kopie des Programms im Speicher halten. OS/2 versieht die entsprechenden Segmente mit der Markierung für internen gemeinsamen Zugriff und gibt sie nicht frei, solange sie noch von mindestens einem Benutzer benötigt werden. Dies entspricht nicht dem gemeinsamen Speicher, wie er üblicherweise unter OS/2 zur Verfügung steht. Da Pure-Segmente nur gelesen und nicht geschrieben werden können, kann kein Prozeß mit Sicherheit wissen, ob sie gleichzeitig von mehreren Prozessen eingesetzt werden. Threads von zwei oder mehr Prozessen können das gemeinsame Pure-Codesegment gleichzeitig ausführen, dies entspricht jedoch nicht einem Multithread-Prozeß. Jede Kopie des Programms besitzt eigene Datenbereiche, einen eigenen Stapel, eigene Dateinummern usw. Die Programme haben im Grunde überhaupt nichts miteinander zu tun, sieht man davon ab, daß OS/2 die gemeinsame Nutzung des Pure-Codesegments erlaubt. Im Unterschied zu mehreren Threads innerhalb eines einzelnen Prozesses können sich Threads unterschiedlicher Prozesse nicht gegenseitig beeinflussen. Für den Programmierer ist die gemeinsame Nutzung von Pure-Codesegmenten bei der Anwendungsentwicklung bedeutungslos.

Da die Pure-Segmente eines Dynlink-Pakets zur gemeinsamen Nutzung zur Verfügung stehen, können alle Klienten nach dem ersten den Ladevorgang wesentlich schneller durchführen. Die Pure-Segmente müssen nicht nochmals aus der .DLL-Datei gelesen werden. Das bedeutet freilich nicht, daß überhaupt kein Dateizugriff nötig wäre. Viele Dynlink-Pakete verwenden Instanzdatensegmente, und OS/2 muß die Ausgangswerte dieser Segmente aus der .DLL-Datei holen.

Der zweite Klient eines Dynlink-Paketes ist der zweite *gleichzeitige* Klient. Unter OS/2 haben lediglich Prozesse eine Eigengültigkeit (*life of their own*). Objekte wie Dynlink-Pakete oder gemeinsam genutzte Speichersegmente existieren nur als Besitztum von Prozessen. Fällt der letzte Klientenprozeß eines derartigen Objektes weg, löscht OS/2 das Objekt und gibt den davon belegten Speicher frei. Wenn zum Beispiel der erste Klient eines Dynlink-Paketes dieses anspricht, lädt OS/2 die Code- und Datensegmente des Paketes. Anschließend ruft das Betriebssystem die Initialisierungsroutine des Pakets auf, sofern es eine gibt. In einer internen Datenstruktur hält OS/2 fest, daß das Dynlink-Paket über einen einzigen Klienten verfügt. Das Hinzukommen weiterer Klientenprozesse wird von OS/2 gezählt. Fällt ein Prozeß weg, verringert sich der Zählerstand. Solange der Zähler nicht auf Null steht, das heißt, solange noch mindestens ein Klient vorhanden ist, existiert das Dynlink-Paket, wobei die Klienten die Globaldatensegmente gemeinsam nutzen können. Fällt der Zählerstand auf Null, entfernt OS/2 Code- und Globaldatensegmente des Dynlink-Paketes aus dem Speicher. Das Paket existiert für das Betriebssystem nicht länger. Benötigt ein neuer Klient das Paket wieder, wird es von OS/2 mit den Daten im Globaldatensegment gemäß .DLL-Vorlage (Originaldaten) erneut geladen.

Für die Codesegmente des Dynlink-Paketes ist diese Handhabung durch das Betriebssystem nicht von Belang. Da es sich um Pure-Segmente handelt (die nur gelesen und nicht verändert werden können), ist es unerheblich, ob die Segmente aus der .DLL-Datei neu geladen werden. Anders verhält es sich mit den Globaldatensegmenten. Diese Segmente werden möglicherweise zunächst von Klientenprozessen mit wichtigen Daten gefüllt. Passiert es nun, daß das Dynlink-Paket für einige Zeit von keinem Prozeß genutzt wird, gehen die Daten im Globaldatensegment verloren. Diese Daten sind auch beim erneuten Laden durch einen Klientenprozeß nicht mehr verfügbar, da das Globaldatensegment mit den Ausgangswerten aus der .DLL-Datei initialisiert wird. Wenn dies ein Problem darstellt, muß das Dynlink-Paket eine Art "Leerlaufprozeß" veranlassen, der nur

die Aufgabe hat, dem Dynlink-Paket ständig als Klient zugeordnet zu sein, so daß das Paket (und damit das Globaldatensegment) auch beim Wegfall aller anderen Klienten im Speicher erhalten bleibt.

Hinweis: Bei dieser Methode ist es wichtig, daß die abgekoppelte Form von DosExec verwendet wird (siehe 7.8).

Eine andere Möglichkeit, die Daten im Globaldatensegment vor dem Verlust zu bewahren, besteht darin, einen Klientenzähler einzurichten, der vor dem Entfernen des Dynlink-Paketes (bei Wegfall des letzten Klienten) die Daten im Segment in einer Datei speichert. Die Implementierung dieser Methode erfordert allerdings einen Trick, weil es passieren kann, daß der letzte Prozeß beendet wird, ohne sich beim Dynlink-Paket am entsprechenden Einsprungspunkt korrekt abzumelden. Das Dynlink-Paket muß die Klientenzählung aus diesem Grund über *DosExitList* vornehmen. Ist der Klientenprozeß an das Dynlink-Modul über *DosLoadModule* angehängt (das heißt, als laufzeitdynamische Verknüpfung), kann er nicht mittels *DosFreeModule* abgekoppelt werden, solange die *DosExitList*-Adresse auf das Dynlink-Paket verweist. Beim Versuch kommt es zur Rückgabe eines Fehlercodes. Nun könnte die Anwendung diesen Fehlercode einfach ignorieren; da das Dynlink-Paket aber nach wie vor mit dem Klientenprozeß verbunden ist, erhält es den *DosExitList*-Service bei Prozeßbeendigung. Wichtig ist, daß ein Dynlink-Paket, das Informationen über die Anzahl der angeschlossenen Klienten benötigt, eine *Beendet*-Funktion bereitstellt, die von zu beendenden Prozessen aufgerufen werden kann. Bei Aufruf der Funktion durch einen Klienten muß das Paket die Prozeßadresse aus *DosExitList* entfernen, damit der Prozeß mit *DosFreeModule* beendet werden kann.

Die Tatsache, daß OS/2 aktive Dynlink-Bibliotheken zur gemeinsamen Benutzung bereitstellt, hat Auswirkungen beim Austausch von Dynlink-Paketen. OS/2 hält die .DLL-Datei offen, solange die Bibliothek noch Klienten hat. Daher müssen vor dem Austausch einer Dynlink-Bibliothek sämtliche Klientenprozesse beendet werden.

Dynlink-Segmente können genau wie .EXE-Dateisegmente durch den Linker als *preload* (vorher zu laden) oder als *load on demand* (bei Bedarf zu laden) markiert werden. Beim Laden einer .EXE-Datei oder eines Dynlink-Moduls werden die mit *preload* versehenen Segmente sofort in den Speicher geholt, nicht jedoch die mit *load on demand* bezeichneten. Hierbei gibt es allerdings eine Ausnahme: Liegt die .EXE-Datei oder das Dynlink-Modul auf einem austauschbaren Datenträger vor (etwa einer Diskette), lädt OS/2 in jedem Fall alle Segmente, also auch die als *load on demand* markierten. Bei einem nichtaustauschbaren Datenträger (etwa einer Festplatte) lädt OS/2 jedoch die mit *load on demand* bezeichneten Segmente erst zu dem Zeitpunkt, an dem sie tatsächlich benötigt werden. Dadurch wird die Zeit zum Laden einer Bibliothek verringert und insbesondere das Auslagern von Segmenten minimiert (da die Anzahl der im Speicher gehaltenen Segmente zumindest potentiell verringert wird). Ist ein Segment einmal geladen, wird die Unterscheidung zwischen *preload* und *load on demand* hinfällig, sie ist insbesondere für die Auslagerung der Segmente ohne Bedeutung.

Ein anderer Aspekt ist ebenfalls noch erwähnenswert: OS/2 achtet darauf, daß keine kreisförmig geschlossenen Dynlink-Verknüpfungen auftreten. Da ein Dynlink-Paket ein anderes Dynlink-Modul aufrufen kann, besteht die Gefahr, daß Paket A Paket B aufruft und umgekehrt Paket A wieder von Paket B aufgerufen wird. Selbst wenn der Paket A aktivierende Klientenprozeß C beendet wird, könnten sich die Pakete A und B gegenseitig aktiv halten, so daß beide im Speicher verblieben. OS/2 legt daher ein Schema an, in dem alle Dynlink-Module und deren Klientenprozesse verzeichnet werden. Kann ein Prozeß ein Dynlink-Paket über dieses Schema nicht mehr erreichen - hat, mit anderen Worten, ein Dynlink-Paket keinen Prozeß für einen Klienten und hat keines seiner Klientenpakete Prozesse für Klienten usw. - so wird das Dynlink-Paket aus dem Speicher entfernt. In Bild 7.10 ist ein Beispiel für einen geschlossenen Kreis von Dynlink-Bezugnahmen zu sehen. PA und

PB sind zwei Prozesse, LA bis LG sind Dynlink-Bibliotheksroutinen.

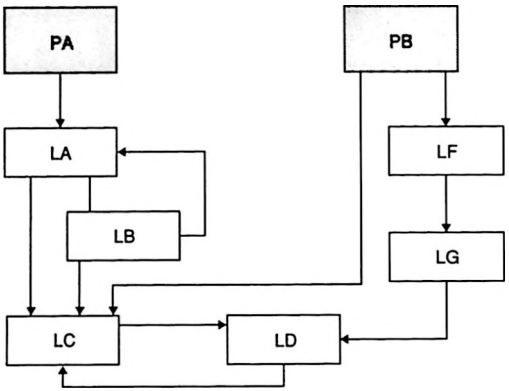


Bild 7.10 Kreisförmige Dynlink-Referenzen

7.11.3 Nebeneffekte von Dynlink-Modulen

Eine sorgsam entwickelte Dynlink-Bibliothek sollte im Idealfall gemäß der Konzeption von OS/2 keinerlei Nebeneffekte hervorrufen. Sie sollte dem Klientenprozeß die festgelegte Funktionalität offerieren und ansonsten überhaupt keine Auswirkungen auf den Klienten haben.

Bei einer Reihe von Aspekten ist dies offensichtlich. So darf eine Dynlink-Routine natürlich keine Dateinummern schließen, die sie nicht geöffnet hat. Das gleiche gilt im Grunde für andere Systemressourcen, auf die der Klientenprozeß zugreift. Dieses Prinzip hat aber auch umgekehrt Gültigkeit: Eine Dynlink-Routine darf Systemressourcen nur unter der Bedingung binden, daß dies keine Auswirkungen auf den Klientenprozeß hat. Wenn eine Dynlink-Routine sehr viele Dateinummern belegt, wäre dies ein unerwünschter Nebeneffekt, da die Dateinummern möglicherweise dem Klienten fehlen. Benötigt eine Dynlink-Routine tatsächlich übermäßig viele Dateinummern, sollte sie beim Betriebssystem die Anzahl möglicher Dateinummern heraufsetzen, so daß noch genügend Nummern für den Klien-

tenprozeß übrigbleiben. Ein anderes Beispiel stellt der Stapel dar. Eine Dynlink-Routine darf nicht so viel Stapelplatz verbrauchen, daß der Klientenprozeß in Schwierigkeiten gerät. Im Gegenteil sollte bei der Entwicklung einer Dynlink-Routine mit dem Stapel besonders vorsichtig umgegangen werden. Wird bei einer neueren Version einer Dynlink-Routine der Stapelverbrauch gegenüber der Vorgängerversion merklich erhöht, kann es leicht passieren, daß die Routine in allen Tests problemlos funktioniert, in der Praxis jedoch nicht korrekt arbeitet, da den angeschlossenen Klientenprozessen zu wenig Stapelkapazität verbleibt.

Eine Dynlink-Routine kann auch durch Systemaufrufe Nebeneffekte verursachen. Da die Dynlink-Routine als Unterroutine des Klientenprozesses abläuft, muß man darauf achten, daß die Systemaufrufe der Routine die Klientenapplikation nicht stören. Ein gutes Beispiel findet sich bei den Signalen. Jedes Signalereignis kann nur eine Adresse für die Signalhandhabung besitzen. Bringt die Dynlink-Routine eine eigene Signalhandhabung ein, steht diese vor jeder Handhabungsroutine des Klientenprozesses. Ein anderes Beispiel: Ändert die Dynlink-Routine die Priorität des Threads, mit dem sie aufgerufen wurde, muß sie dafür sorgen, daß die Ursprungspriorität vor der Rückkehr zum aufrufenden Prozeß wiederhergestellt wird. Diverse andere Systemfunktionen wie *DosError* oder *DosSetVerify* führen ebenfalls zu Nebeneffekten, die sich auf den Klientenprozeß auswirken können.

Es ist unmöglich, alle Eventualitäten aufzuzählen, bei denen es zu Nebeneffekten kommen kann. Es ist Aufgabe des Entwicklers eines Dynlink-Moduls, das Paket in einer möglichst praxisnahen, das heißt einsatznahen, Systemumgebung zu testen. Die oben geschilderten Nebeneffekte oder ähnliche sollten unter allen Umständen vermieden werden. Fast noch wichtiger ist, daß ein Dynlink-Modul in seinem Verhalten konsistent ist. Nur dadurch kann erreicht werden, daß ein ausgetestetes Paket in der Praxis nicht "aus heiterem Himmel" Fehler hervorruft. Das gilt noch stärker für neue Versionen existierender Dynlink-Module. Bei allen Entwicklungsanstrengungen ist darauf zu achten, daß

ein Klient, der für die Erstversion eines Dynlink-Paketes geschrieben ist, auch mit der jeweils neuesten Version problemlos läuft. Der Anwendungsentwickler, der die Verantwortung für den Klientenprozeß trägt, hat keine Möglichkeit, die Verträglichkeit seiner Applikation mit einer neuen Dynlink-Routine vorab zu testen.

7.12 Dynlink-Namen

Jedem Dynlink-Einstiegspunkt sind drei Namen zugeordnet: der externe Name, der Modulname und der Name des Einsprungspunktes. Der externe Name ist der Name, unter dem der Klientenprozeß die Funktion aufruft. Der Anwendungsprogrammierer arbeitet mit diesem Namen, der in Syntax und Form zum verwendeten Assembler oder Compiler passen muß. Der Name sollte einfach und möglichst selbsterklärend sein. Gleichzeitig ist jedoch ein Name zu wählen, bei dem es nicht zur Kollision mit anderen Namen kommt. READ oder RESET ist beispielsweise keine gute Namenswahl, weil man davon ausgehen muß, daß diese Namen in diversen Modulen verwendet werden. Andererseits stellt es eine Zumutung dar, mit einem Namen wie XR23P11 umgehen zu müssen.

Der Linker ersetzt den externen Namen durch einen Modul- und einen Einstiegspunktnamen, die Teile der erzeugten .EXE-Datei werden. OS/2 identifiziert die .DLL-Datei anhand des Modulnamens, der als Dateiname mit der Dateinamenserweiterung .DLL verwendet wird. Der Einstiegspunktname schließlich stellt den Bezug zum Einsprungspunkt des Paketes her. Dieser Name muß nicht mit dem externen Namen übereinstimmen. Bei Modulen mit vielen Einstiegspunkten empfiehlt es sich, anstelle der Einstiegspunktnamen Zahlen zu verwenden. Das reduziert den Umfang der .EXE-Datei und erhöht die Geschwindigkeit beim Laden des Dynlink-Moduls. In den technischen Unterlagen zu OS/2 finden Sie Einzelheiten zu diesem Thema.

Laufzeitdynamische Verknüpfungen werden durch den Modul- und die Einstiegspunktnamen hergestellt; der externe Name wird hier nicht verwendet.

Kapitel 8

Dateiverwaltung

Die Dateiverwaltungssysteme von OS/2 und MS-DOS unterscheiden sich voneinander.

In OS/2 wird ein einheitliches Schema zur Vergabe von Namen im ASCII-Format für Objekte wie Dateien, Semaphoren, gemeinsame Speicherbereiche usw. verwendet. Dieses Konzept wird im folgenden erläutert.

8.1 Dateinamen

Bevor die Namensvergabe unter OS/2 erklärt wird, soll das Format der Dateinamen unter MS-DOS angesprochen werden. Unter MS-DOS besitzen Dateinamen das 8.3-Format, mit einem Namensfeld (maximal acht Zeichen) und einem Erweiterungsfeld (Dateinamenergänzung aus maximal drei Zeichen). Der Punkt zwischen dem Namen und der Erweiterung ist nicht Teil des Dateinamens, sondern ein Begrenzungszeichen. Der Dateiname kann ausschließlich Großbuchstaben beinhalten. Bei dem Versuch, einen Dateinamen anzulegen, der Kleinbuchstaben oder eine Mischung aus Klein- und Großbuchstaben enthält, konvertiert MS-DOS den Dateinamen vollständig in Großbuchstaben. Zu lange Dateinamen werden von MS-DOS abgeschnitten. Das Betriebssystem achtet auf die strenge Einhaltung der Regeln und bestimmt dabei auch die Struktur auf der Diskette bzw. Platte. Das Dateiverwaltungssystem, das von MS-DOS Version 3.X unterstützt wird, ist die Dateibelegungstabelle (FAT für *file allocation table*). Im folgenden finden Sie typische MS-DOS- und OS/2-Dateinamen:

```
\FUSSBALL\SRC\KERNEL\SCHED.ASM
```

Hinweis: Die Festlegungen bezüglich des Umfangs von Dateinamen gehen auf die Digital Equipment Corporation (DEC) zurück. Die ersten von DEC konstruierten Computer verwendeten die sog. RAD50-Technik zur Speicherung von drei Großbuchstaben in einem 16-bit-Wort. Daher bot es sich an, im Dateiverwaltungssystem nur Dateinamen aus sechs Zeichen und Namenerweiterungen aus drei Zeichen zu erlauben. CP/M hat diese Dateinamenstruktur übernommen. Da CP/M aber nicht RAD50 als Format verwendete, durfte der Dateiname "großzügigerweise" sogar acht Zeichen umfassen; die drei Zeichen lange Erweiterung blieb erhalten.

Fussball steht hier stellvertretend für ein Projekt. Der Pfadname führt zum Zeitplan (Scheduler) für das Projekt.

```
\MEMOS\286\MODUS.TXT
```

In diesem Memo werden die Modi des 80286 behandelt.

```
\\HAGAR\ZWISCHEN\GORDON\FUERMARK
```

Der Name beschreibt eine Datei in einem Zwischenspeicherverzeichnis auf dem Netzwerk-Server HAGAR, die dort vom Benutzer *Mark* angelegt wurde.

Unter OS/2 sieht die Dateiverwaltung anders aus. Für Mikrocomputer in der Leistungsklasse, die zunehmend verfügbar wird, ist das geschilderte einfache Dateiverwaltungssystem unzureichend. Hier ist beispielsweise an Peripheriegeräte wie WORM-Laufwerke (*write once read many*) zu denken, die eine spezielle Dateiverwaltung benötigen. Aus diesem Grund ist das Dateiverwaltungssystem in OS/2 nicht fester Bestandteil des Betriebssystems. Vielmehr verfügt OS/2 über ein installierbares Dateiverwaltungssystem (IFS für *installable file system*). Ein IFS ähnelt einem Einheitentreiber, der entsprechende Programmcode wird beim Start von OS/2 geladen. Die Kommunikation zwischen dem IFS und OS/2 erfolgt über die Standardschnittstelle; das Dateiverwaltungssystem stellt die Software zur Verwaltung des Dateisystems der Speichermedien einschließlich der Möglichkeiten zum Anlegen und Warten von Verzeichnissen, Belegen von Diskettenkapazität usw. bereit.

Hinweis: WORM-Laufwerke enthalten im allgemeinen Laserplatten mit hoher Kapazität. Eine einmal beschriebene Spur kann nicht mehr gelöscht werden. Dennoch läßt sich der Eindruck der Löschbarkeit erwecken, indem neue Versionen von Dateien und Verzeichnissen nach einer Veränderung erneut aufgezeichnet und die alten Versionen abgekoppelt werden.

Sofern Sie mit OS/2 Version 1.0 vertraut sind, wird Ihnen das IFS-Konzept seltsam vorkommen, da es im Handbuch keine Erwähnung findet. Der Grund: Die Implementierung hinkt der architektonischen Konzeption zeitlich hinterher. OS/2 wurde jedoch von Anfang an auf die Unterstützung installierbarer Dateiverwaltungssysteme hin ausgelegt, von denen eines das bekannte FAT-Dateiverwaltungssystem sein kann. Dateiverwaltungssystemaufrufe wie *DosOpen* und *DosClose* wurden vor diesem Hintergrund implementiert. Die strenge Zeitplanung und der daraus resultierende Druck veranlaßten uns, OS/2 Version 1.0 ausschließlich mit einem FAT-Dateisystem ausgestattet auszuliefern - eine künftige Version wird das vollständige IFS-Paket enthalten. Zum Zeitpunkt, da das vorliegende Buch verfaßt wurde, liegt die offizielle Ankündigung von IFS noch in der Zukunft. Informationen hierüber sind aber für die Anwendungsentwicklung unter OS//2 notwendig und sollen daher im folgenden gegeben werden.

Da das IFS Datei- und Pfadnamen interpretieren wird und installierbare Dateiverwaltungssysteme große Unterschiede aufweisen können, enthält OS/2 selbst (abgesehen vom IFS) keine Informationen über das Format und die Bedeutung von Datei- und Pfadnamen. Das Datei- und Pfadnamenformat wird vom IFS selbständig verwaltet, sowohl das Anwendungsprogramm als auch OS/2 stellen Zwischenschritte dar, denen keine diesbezüglichen Informationen zur Verfügung gestellt werden müssen. Sie sollten nicht den Versuch unternehmen, Datei- und Pfadnamen zu zerlegen, da das IFS auch Namen in anderen Formaten als dem 8.3-Format von MS-DOS unterstützt. Selbst auf eine feste Länge für einen Datei- oder Pfadnamen kann man sich nicht verlassen. Alle OS/2-Datei- und Pfadnamenschnittstellen (wie *DosOpen* oder *DosFindNext*) wurden auf die Übernahme von Zeichenketten beliebiger Länge ausgelegt. Anwendungsprogramme sollten Namenpuffer mit einer Länge von mindestens 256 Zeichen verwenden, damit sichergestellt ist, daß lange Dateinamen nicht abgeschnitten werden.

8.2 Netzwerkzugriff

Eine Länge von 256 Zeichen zur Speicherung eines Dateinamens erscheint auf den ersten Blick etwas übertrieben. Andererseits enthalten OS/2-Dateinamen oft Pfadnamen, die sehr lang werden können. Um einen transparenten Zugriff auf Dateien in einem LAN (*local area network*) zu erlangen, bindet OS/2 das Netzwerk als Teil des Dateiverwaltungssystems ein. Mit anderen Worten: Ein Pfadname kann neben einem Verzeichnispfad auch einen Gerätenamen enthalten. Ein Anwendungsprogramm könnte einen Öffnungsaufruf mit einem Namenstring wie \ARBEIT-\BUCH.DAT oder \\VOGON\TEMP\NEUBER.ASM ausgeben. Der erste Name spezifiziert die Datei BUCH.DAT im Verzeichnis ARBEIT des aktuellen Laufwerks des lokalen Gerätes. Der zweite Name spezifiziert die Datei NEUBER.ASM im Verzeichnis TEMP des Gerätes VOGON. Künftige Versionen des Microsoft LAN Manager werden das Dateiverwaltungssystem von OS/2 nutzen, so daß Dateinamen (insbesondere programmgenerierte) schnell sehr lang werden können.

Hinweis: Die Benennung von Netzwerken ist komplizierter: Der Name TEMP des Gerätes VOGON bezieht sich auf die verfügbare Netzwerk-Ressource und kann in einem beliebigen aktuellen Verzeichnis stehen.

8.3 Namensgenerierung und Kompatibilität

Anwendungsprogramme sollten idealerweise das Format der vom Benutzer eingegebenen Dateinamen außer acht lassen. Dies ist natürlich unrealistisch. Programme müssen oftmals selbst Dateinamen generieren (zur Zwischenspeicherung von Daten, zur Aufnahme abgeleiteter Dateinamen usw.). Wie kann eine Anwendung Dateinamen anlegen oder verändern und dennoch die Kompatibilität mit allen installierbaren Dateiverwaltungssystemen gewährleisten? Die Antwort: Verwenden Sie den ältesten Standard, von dem Sie sicher annehmen können, daß er von allen Systemen unterstützt wird. Ein neues IFS wird FAT-Dateinamen (im 8.3-Format) akzeptieren, da es andernfalls mit vielen anderen Programmen inkompatibel wäre. Wenn ein Anwendungsprogramm sich auf die 8.3-Formatregel beim Anlegen von Dateinamen beschränkt, ist die Kompatibilität mit künftigen

Dateiverwaltungssystemen sichergestellt. Im Gegensatz zu MS-DOS schneidet OS/2 (genauer: das Dateiverwaltungssystem von OS/2) den Dateinamen oder die Erweiterung nicht ab, wenn diese zu lang sind; statt dessen wird ein Fehler gemeldet. Die Groß-/Kleinschreibung eines Dateinamens wird auch weiterhin ohne Signifikanz bleiben. Einige Betriebssysteme wie UNIX unterscheiden bei Dateinamen zwischen Groß- und Kleinbuchstaben. Die Namen *Arbeit* und *arbeit* bezeichnen in Unix unterschiedliche Dateien. Diese Methode funktioniert, solange ein System durchweg von Programmierern genutzt wird. Für einen Softwareentwickler ist es selbstverständlich, daß es sich bei dem Kleinbuchstaben *f* (ASCII-Code 66 hex) und dem Großbuchstaben *F* (ASCII-Code 46 hex) um unterschiedliche Zeichen handelt.

Für einen unbedarften Benutzer hingegen bezeichnen *f* und *F* den gleichen Buchstaben. Da der Großteil der OS/2-Anwender keine Programmierer sein werden, wird das installierbare Dateiverwaltungssystem von OS/2 keine Unterscheidung bezüglich der Groß-/Kleinschreibung vornehmen.

Wie bereits ausgeführt, sollten programmgenerierte Namen der 8.3-Formatregel folgen, um Komplikationen von vornherein auszuschließen. Ebenfalls aus Sicherheitsgründen empfiehlt es sich, Programmnamen nur derart zu modifizieren, daß alphanumerische gegen andere alphanumerische Zeichen ausgetauscht werden (zum Beispiel: ARBEIT.OBJ gegen ARBEIT.ASM). Die Verlängerung von Dateinamen ist auch möglich (zum Beispiel ARBEIT.C zu ARBEIT.OBJ), wenn das Programm den neuen Namen auf seine Gültigkeit hin untersucht. Falls sich der Name als ungültig erweisen sollte, muß eine Korrekturmöglichkeit zur Verfügung stehen. In jedem Fall sollte das Programm erweiterte Dateinamen verarbeiten können. In den oben genannten Substitutionsfällen ist es ratsam, daß der Algorithmus vom Ende der Zeichenkette her arbeitet.

8.4 Berechtigungen

In künftigen OS/2-Versionen wird das Dateiverwaltungssystem nicht nur zur Bezeichnung von Dateien verwendet werden können, sondern auch zur Vergabe von Zugriffsberechtigungen. Ein Algorithmus wird eine Zugriffsliste mit jedem Eintrag im Dateinamensbereich verknüpfen, so daß der unautorisierte Zugriff (versehentlich oder vorsätzlich) auf die genannte Datei unterbunden wird.

8.5 Andere Objekte im Dateinamensbereich

Die Bedeutung des Dateinamensbereiches wurde bereits anhand mehrerer Aspekte beleuchtet. Zum einen erlaubt der Bereich die Benutzung einer Vielzahl unterschiedlicher Namen. Namen können gruppiert werden (durch das Speichern im gleichen Verzeichnis) und es lassen sich Familien von einheitlichen Namen (durch das Anlegen neuer Unterverzeichnisse) anlegen. Zum zweiten kann der Namensbereich alle Dateien und Einheiten des lokalen Rechners ebenso wie die Dateien und Einheiten von Remote-Systemen umfassen. Schließlich werden Dateisystemnamen künftig einen flexiblen Zugriffsschutz unterstützen.

Es ist keine Überraschung, daß die Entwickler bei der Suche nach Methoden für die Benennung von Objekten, die keine Dateien darstellen (gemeinsam benutzter Speicherbereich, Systemsemaphoren und benannte Pipes), auf den Dateisystemnamensbereich zurückgegriffen haben. Ein Nachteil dieser Entscheidung ist darin zu sehen, daß Dateien und Objekte, die keine Dateien darstellen, unterschiedliche Namen aufweisen müssen. Ein vergebener Dateiname kann nicht mehr für einen gemeinsam genutzten Speicherbereich, eine Systemsemaphore oder eine benannte Pipe verwendet werden, was allerdings verglichen mit den Vorteilen der gemeinsamen Benutzung des Dateisystemnamensbereiches kaum relevant ist. Zudem können für jeden Objekttyp eigene Unterverzeichnisse verwendet werden, so daß Überlappungen von vornherein ausgeschlossen sind.

Dies bedeutet übrigens nicht, daß Systemsemaphoren, gemeinsam genutzter Speicher und Pipes Dateisystemeinträge auf einer Diskette/Festplatte besitzen. Das FAT-Dateisystem unterstützt in seinen Verzeichnissen keine Sonderobjektnamen. Eine Veränderung in dieser Richtung wäre leicht möglich, doch wäre das FAT-Dateisystem dann nicht mehr abwärtskompatibel (MS-DOS 3.X könnte Disketten/Platten, die unter OS/2 beschrieben wurden, nicht lesen). Da für OS/2 Version 1.0 nur das FAT-Dateiverwaltungssystem verfügbar ist, enthält diese Version RAM-residente Pseudoverzeichnisse, die Objektnamen aufnehmen. Die Namen müssen mit \SEM\, \SHAREMEM\, \QUEUES\ und \DEV\ beginnen, um die Gefahr der Namenskollision mit Dateien in künftigen Versionen von OS/2 zu minimieren.

Obgleich Leistungsmerkmale wie Netzwerkbetrieb und (zukünftig) Zugriffsschutz zum Dateisystemnamensbereich gehören (vom Standpunkt der Architektur her), werden nicht alle Permutationen unterstützt. Insbesondere die Unterstützung benannten gemeinsam benutzten Speicherbereiches über ein Netzwerk ist nur mit sehr viel Aufwand zu realisieren. Dies hängt damit zusammen, daß das gesamte Speichersegment bei jeder Änderung im Netzwerk übertragen werden muß. Durch eine Reihe von Optimierungsmaßnahmen läßt sich der Aufwand reduzieren, was aber immer noch nicht zu einem vernünftigen Aufwand/Nutzen-Verhältnis führt. Dieses Leistungsmerkmal wird daher auch künftig in OS/2 nicht implementiert werden.

Kapitel 9

Speicherverwaltung

Die primäre Funktion eines Multitasking-Betriebssystems besteht in der Belegung von Systemressourcen für die einzelnen Prozesse. Der Scheduler teilt die CPU-Zeit unter den Prozessen bzw. im engeren Sinne unter den Threads auf; der *Memory Manager* belegt sowohl den physikalischen als auch den virtuellen Speicher.

9.1 Schutzmodell

OS/2 arbeitet im Unterschied zu MS-DOS mit einer Speicherverwaltung, die den Schutz von Speicherbereichen vorsieht (*Protect*-Modus). Unter MS-DOS 3.X *sollte* ein Programm vom Betriebssystem die Zuweisung eines Speicherbereiches anfordern. Unter OS/2 *muß* ein Programm das Betriebssystem zur Speicherbereichsbelegung auffordern. Wie bereits erläutert, besitzt der Mikroprozessor 80286 eine Hardware-Vorrichtung zum Schutz von Speicherbereichen. Jede Bezugnahme auf einen Speicherbereich stellt eine explizite oder implizite Referenz auf einen Segmentselektor dar. Der Segmentselektor wiederum bezieht sich auf einen Eintrag in der GDT oder LDT (abhängig von der Art des Selektors). Bevor ein Programm (einschließlich OS/2 selbst) eine Speicherstelle ansprechen kann, muß die Speicherposition in einem LDT- oder GDT-Eintrag beschrieben werden. Der Selektor für den Eintrag muß in eines der vier Segmentregister geladen werden.

Diese Hardware-Konzeption verursacht bei der Verwendung von Speicheradressen durch Programme einige Einschränkungen:

- Ein Programm kann keinen Speicherbereich ansprechen, der nicht in der LDT oder GDT beschrieben ist. Die einzige Möglichkeit zur Speicheradressierung ist der Weg über eine LDT und GDT.
- Jeder Segmentdeskriptor in der LDT oder GDT enthält die physikalische Adresse und Länge des Segmentes. Ein Programm kann keinen *Offset* im Segment ansprechen, der die Länge des Segmentes überschreiten würde.
- Ein Programm kann keine willkürlichen Werte in ein Segmentregister schreiben. Beim Laden eines Segmentregisters untersucht die Hardware die zugehörige LDT und GDT und überprüft, ob der Eintrag gültig ist. Falls vom Programm ein willkürlicher Wert - beispielsweise die niederwertige Hälfte einer Fließkommazahl - in ein Segmentregister geschrieben wird, zeigt dieser Wert - vermutlich - auf einen ungültigen LDT- oder GDT-Eintrag und ruft damit einen GP-Fehler hervor.
- Ein Programm kann keine Befehle ausführen, die sich in einem Datensegment befinden. Beim Versuch, einen Datensegmentsektor in das Register CS zu laden (üblicherweise bei einem *far*-Aufruf oder -Sprung), kommt es zu einem GP-Fehler.
- Ein Programm kann nicht in das Codesegment schreiben. Der Versuch führt zu einem GP-Fehler.
- Ein Programm kann keine *Segmentarithmetik* ausführen. Die Segmentarithmetik bezieht sich auf den Adressierungsmechanismus des 8086/8088. Obwohl von diesen beiden Prozessoren gesagt wird, daß sie eine Segmentarchitektur besitzen, sind sie im Grunde linear adressierende CPUs, die *Offset*-Register (sogenannte *Segmentregister*) einsetzen. Der 8086 kann 1 Mbyte Speicherbereich adressieren und benötigt dazu eine 20-bit-Adresse. Der Prozessor legt die Adresse durch die Multiplikation eines 16-bit-Segmentwer-

tes mit 16 und dem Addieren eines 16-bit-Offsetwertes fest. Das Resultat ist eine Adresse, die zwischen 0 und 1.048.575 (1 Mbyte) liegt. Man sollte hier nicht von Segmenten sprechen, da die zugehörigen Längenangaben fehlen und die Selektoren keine Namen aufweisen. Vielmehr handelt es sich bei diesen "Segmenten" um physikalische Adressen, die durch den Wert 16 dividiert wurden. Die Segmentwerte sind skalierte Offsetwerte. Die Adresse mit dem Segmentwert 100 und dem Offsetwert 100 (geschrieben als 100_{10}:100_{10}) und die Adresse 99_{10}:116_{10} beziehen sich auf die gleiche Speicherstelle.

Hinweis: Die höchste, auf diese Weise ansprechbare Adresse liegt bei über einem Mbyte (2^{20}), wenn Segment- und Offsetwert groß genug gewählt werden. Der 8086 übersieht den Übertrag in das nicht existierende 21. Adressenbit und es wird der Anfang des physikalischen Speichers angesprochen.

Bei im *Real*-Modus ablaufender Software werden aus dieser Situation oftmals Vorteile gezogen. Programme mit vielen Zeigern speichern diese als 20-bit-Werte, die lediglich in die Form *Segment:Offset* zerlegt werden, wenn dies nötig wird. In anderen Programmen wird zur Sicherstellung eines spezifischen Offsetwertes für bestimmte Objekte ein entsprechender Segmentwert gewählt, so daß die abgeleitete 20-bit-Adresse korrekt ist. Diese Methoden funktionieren jedoch nicht im *Protect*-Modus von OS/2. Jeder Segmentselektor beschreibt ein eigenes Segment mit einer Länge und einer Adresse, die vom numerischen Wert des Segmentselektors unabhängig sind. Der durch das Segment *N* spezifizierte Speicherbereich hat mit dem durch das Segment *N+4* beschriebenen Speicherbereich nichts gemein. Gleiches gilt für alle anderen Segmente, solange OS/2 keine explizite Verknüpfung anlegt.

Durch die Segmentierung und die dem Schutz dienende Hardware muß OS/2 den Prozessen weitere Beschränkungen auferlegen:

- Prozesse können den Inhalt der LDT oder GDT weder untersuchen noch verändern. OS/2 verweigert einfach die Erstellung von LDT- oder GDT-Selektoren, die ein Prozeß

zum Zugriff auf den Tabelleninhalt verwenden könnte. Es gibt LDT- und GDT-Selektoren, die auf den Inhalt der Tabelle verweisen; OS/2 initialisiert diese aber so, daß sie lediglich von Ring-0-Code verwendet werden können, das heißt von Programmen, die im *Privilege*-Modus ablaufen.

- Interruptvektoren können von Prozessen nicht verändert werden. Programme unter MS-DOS Version 3.X ändern Interruptvektoren üblicherweise, indem die Adresse der interruptbearbeitenden Routine durch die Adresse eines selbstgeschriebenen Programms ersetzt wird. Die Programme können Systemaufrufe über INT 21H, BIOS-Interruptaufrufe und Hardwareinterrupts (Tastatur und Systemuhr) überwachen. OS/2-Programme sind dazu nicht in der Lage. OS/2 erlaubt die Initialisierung eines Segmentselektors nicht, mit dem Prozesse die Interruptvektortabelle adressieren könnten.

- Prozesse können das ROM-BIOS nicht ansprechen, da kein Selektor den ROM-BIOS-Programmcode adressiert. Ein solcher Selektor wäre allerdings auch nur von geringem Nutzen. Das ROM-BIOS wurde für die Ausführung im *Real*-Modus entwickelt und führt Segmentarithmetikoperationen aus, die unter OS/2 keine Gültigkeit besitzen. Hätte OS/2 einen ROM-BIOS-Selektor, würden Aufrufe des ROM-BIOS üblicherweise zu GP-Fehlern führen.

- Prozesse können nicht im Ring-0-Modus (*Privilege*-Modus) ausgeführt werden. Diese Beschränkung wurde bewußt sowohl in OS/2 als auch in den Prozessor 80286 eingebaut. Programme mit Ring-0-Status können die LDT- und GDT-Tabellen und andere Hardware-Schutzleistungsmerkmale manipulieren. Falls OS/2 einem Prozeß die Ausführung in Ring 0 erlauben würde, könnte das System niemals stabil bzw. sicher sein. OS/2 selbst arbeitet mit *Privilege*-Status, da es das erste geladene Programm ist. Der Startprozeß findet in Ring 0 statt und übergibt die Ring-0-Berechtigung an OS/2, indem die Kontrolle an OS/2 übertragen wird. Dieses Privileg wird von OS/2 nicht an Anwendungs-

programme weitergegeben. Dadurch ist sichergestellt, daß Anwendungen nur im Anwendungsmodus (in Ring 3) ausgeführt werden können (unter bestimmten Bedingungen können Anwendungsprogramme auch in Ring 2 ausgeführt werden, siehe 18.1).

9.2 Schnittstelle (API) zur Speicherverwaltung

OS/2 bietet eine umfangreiche Schnittstelle für Anwendungsprogramme (*application program interface*, API) zur Speicherverwaltung. Das vorliegende Buch soll kein Handbuch darstellen, aus diesem Grund werden hier nicht alle Aufrufe behandelt, sondern nur ausgewählte.

9.2.1 Gemeinsam benutzter Speicher

OS/2 unterstützt zwei Arten des gemeinsam verwendeten Speichers - den *benannten gemeinsam benutzten Speicher* (*named shared memory*) und den *Übergabespeicher* (*giveaway shared memory*). In beiden Fällen ist der gemeinsam benutzte Speicherbereich ein Segment. Es kann stets nur ein vollständiges Segment gemeinsam angesprochen werden, nicht ein Segmentteil. Weder der Name des gemeinsam benutzten Speichers noch der Speicher selbst können in einem FAT-Dateisystem festgehalten werden. Wenn kein Prozeß das gemeinsame Speichersegment verwendet, wird der Speicherbereich freigegeben (man spricht auch von *flüchtigem Speicher*). Gemeinsam verwendeter Speicher kann nicht ohne Klientenprozeß bestehen, daher muß er mit dem *DosAllocShrSeg*-Aufruf neu initialisiert werden, falls er zwischenzeitlich nicht benutzt wurde.

Der *Übergabespeicher* erlaubt Prozessen den gemeinsamen Zugriff auf dasselbe Segment. Übergabespeicher besitzen keine Namen, da Prozesse den Zugriff darauf nicht anfordern können. Vielmehr muß der aktuelle Benutzer des Segmentes den Zugriff auf das Segment einem anderen Klientenprozeß übertragen. Der Ausdruck *Übergabespeicher* ist nicht ganz richtig, da der vergebende Prozeß die Zugriffsmöglichkeit auf den Speicherbereich behält - der Zugriff wird temporär vergeben, aber nicht end-

gültig "übergeben". Die Verwendung von Übergabespeicher ist nicht so unkompliziert wie die von benanntem gemeinsamem Speicher. Der Besitzer (genauer: einer der Besitzer - jeder Besitzer mit Zugriff auf ein Übergabespeichersegment kann die Zugriffsmöglichkeit weitergeben) muß die PID eines Prozesses kennen und mit dessen Segmentselektor (wird von *DosGiveSeg* zurückgegeben) über IPC kommunizieren.

Trotz seiner Beschränkungen bietet der Übergabespeicher einige wichtige Leistungsmerkmale. Er eignet sich ideal für den schnellen und effizienten Transfer von Daten zwischen Prozessen. Da der Zugriff unmittelbar zwischen zwei Prozessen ("von Hand zu Hand") weitergegeben wird, kann kein Prozeß irrtümlicherweise auf das Segment zugreifen. Die meisten Prozesse behalten den Zugriff auf ein bereits weitergegebenes Segment nicht. Sie führen üblicherweise einen *DosFreeSeg*-Aufruf mit der Nummer auf, die von *DosGiveSeg* bereitgestellt wurde. Hier ist beispielsweise an ein Datenbank-Dynlink-Subsystem zu denken, das die Funktion eines Front-Ends für einen Datenbank-Server-Prozeß übernimmt. Als Teil des *Dynlink*-Initialisierungsprozesses wird ein benanntes Speichersegment eingerichtet, das alle Klientenprozesse gemeinsam benutzen. Eine benannte Pipe oder ein benannter gemeinsam verwendeter Speicherbereich (vom Datenbankprozeß angelegt) sollte zur erstmaligen Kommunikation eingerichtet werden. Diese Schnittstelle dient anschließend zum Anlegen eines Übergabespeicherbereiches für alle weiteren Transaktionen zwischen dem Klientenprozeß (über das *Dynlink*-Subsystem) und dem Datenbankprozeß. Die Methode bietet eine größere Sicherheit als die Aufrechterhaltung eines einzelnen benannten gemeinsam verwendeten Segmentes, das alle Serviceanforderungen aller Klientenprozesse verwaltet. Da jeder Klientenprozeß einen eigenen gemeinsam verwendeten Kommunikationsbereich unterhält, kann kein anderer Prozeß in diese Verknüpfung eindringen.

Die Anforderung des Klientenprozesses an die Datenbank zum Lesen eines Datensatzes führt dazu, daß der Datenbankprozeß eine IPC zur Übertragung der Daten an den Klientenprozeß benötigt. Pipes sind für solche Datenmengen zu langsam, der ge-

meinsam benutzte Speicherbereich bietet sich hier an. Bei der Verwendung gemeinsam benutzten Speichers muß das Datenbankpaket für jeden Datensatz einen eigenständigen gemeinsam benutzten Speichernamen anlegen, Speicher reservieren und den Namen an den Klienten übergeben (eigentlich an das *Dynlink*-Subsystem, das vom Klienten aufgerufen wurde). Nur dann kann der Zugriff angefordert werden. Dieser Prozeß hat einige Nachteile:

- Für jede Anfrage muß ein neuer einzigartiger, gemeinsam benutzter Speichername angelegt werden. Durch die wiederholte Benutzung eines Speichersegmentes würde der Klientenprozeß gezwungen, sich die Daten aus dem Segment zu holen, bevor eine andere Anforderung erfolgen kann. Für Prozesse, die hohe Datenvolumina übertragen, ist das unrealistisch.
- Das Anlegen von benannten Speichersegmenten ist grundsätzlich langsamer als das Anlegen von Übergabespeichersegmenten. Das gilt insbesondere dann, wenn eine große Anzahl benannter gemeinsam benutzter Speichersegmente im System vorliegt. Der Klientenprozeß benötigt mehr Zeit für einen Zugriff auf das Segment. Der Weg über gemeinsam benutzte Speichersegmente ist für viele Anwendungen durchaus schnell genug. In einer zeitkritischen Anwendung wie im vorliegenden Beispiel würde die Datenübertragung aber rasch zum "Flaschenhals" werden.

Statt dessen kann der Datenbankprozeß ein Übergabespeichersegment anlegen, Daten darin speichern und dem Klientenprozeß den Zugriff erlauben. Der Datenbankprozeß kann die PID des Klientenprozesses erfahren. Die *Dynlink*-Schnittstelle, die als Klientenprozeß ausgeführt wird, enthält diese Information als Teil der Datenanforderung. Auf die gleiche Weise kann der Datenbankprozeß den neuen Klientenselektor an den Klienten zurückgeben. Der Prozeß ist schnell und effizient und zwingt das System nicht zur Behandlung umfangreicher Namenszeichenketten.

Beachten Sie, daß bei einem *DosAllocSeg*-Aufruf das Segment als *Übergabesegment* gekennzeichnet werden muß. Dies ermöglicht OS/2 die Ablage des Selektors im reservierten LDT-Bereich.

9.2.2 Große Speichersegmente (Huge Memory)

Die Architektur des Mikroprozessors 80286 legt die maximale Größe eines Speichersegmentes auf 64 Kbyte fest. Für viele Programme ist dieser Segmentumfang zu gering. So beansprucht beispielsweise eine umfangreiche Kalkulationstabelle 256 Kbyte Speicherkapazität oder mehr. OS/2 ist aufgrund der Prozessorbeschränkungen nicht in der Lage, ein Segment größer als 64 Kbyte anzulegen, es verfügt aber über das Leistungsmerkmal des *Huge Segment*, das die Emulation eines größeren Segmentes erlaubt. Der Trick besteht darin, daß sich ein 200 Kbyte umfassendes *Huge Segment* aus mehreren Segmenten zusammensetzt, in diesem Fall aus drei 64-Kbyte-Segmenten und einem 8-Kbyte-Segment. Mit minimalem Programmieraufwand erlaubt OS/2 die Behandlung des *Huge Segment* als einzelnes Segment.

Bei der Ausführung eines *DosAllocHuge*-Aufrufs zur Belegung eines *Huge Segment* reserviert OS/2 mehrere physikalische Segmente, deren Summe der Größe des virtuellen *Huge Segment* entspricht. Alle Komponentensegmente besitzen eine Größe von 64 Kbyte mit der Ausnahme des letzten Segments, das gegebenenfalls weniger Speicherkapazität umfaßt. Im Gegensatz zu einer willkürlichen Zusammenstellung von Segmentselektoren stellt *DosAllocHuge* sicher, daß die zurückgegebenen Selektoren der Segmente einheitlich weit auseinanderliegen. Der Selektor des (*N+1*)-Komponentensegmentes ist der des N. Segmentes + *I*, wobei *I* ein Exponent zur Basis 2 ist. Der Wert von *I* ist für die Ausführung unter OS/2 konstant, kann aber zwischen den verschiedenen OS/2-Versionen variieren. Die interne Konfiguration während des Startvorgangs nimmt ebenfalls Einfluß auf den Wert *I*. Ein Programm muß den Faktor *I* bei jeder Ausführung erneut bestimmen, da es sich nicht um einen starr festgelegten Wert handelt. Es gibt drei Möglichkeiten zur Bestimmung des Wertes. Das Programm kann einen *DosGetHugeShift*-Aufruf

ausführen, den Wert aus dem globalen Informationssegment lesen oder als undefinierte externe Referenz über DOSHUGESHIFT($\log_2(i)$) oder DOSHUGEINCR(i) ansprechen. OS/2 wird den Wert für die externe Referenz beim Laden des Programms korrekt ersetzen. Die letztgenannte Methode ist am effizientesten und somit empfehlenswert, sie ist aber nicht mit der *Family API* kompatibel. API-Programme sollten einen *DosGetHugeShift*-Aufruf ausführen. In Bild 9.1 wird die Belegung eines 200-Kbyte-*Huge*-Speicherobjektes gezeigt. Die Selektoren $n + 4i$ und $n + 5i$ sind momentan nicht gültig, aber für den künftigen Zuwachs des *Huge*-Objektes reserviert.

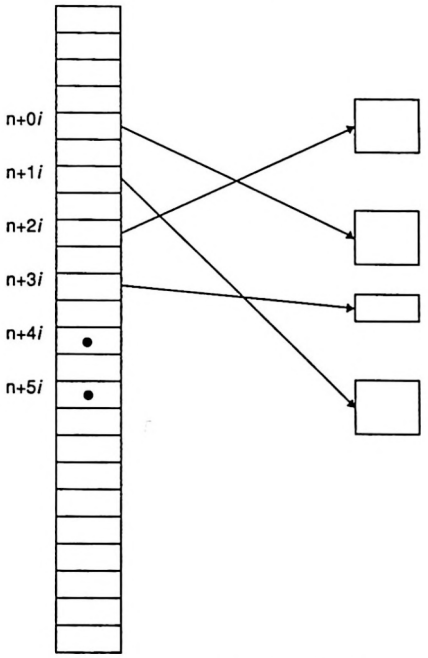


Bild 9.1 Huge-Speicherobjekte

Nachdem eine Anwendung den ersten Segmentselektor einer Gruppe von *Huge*-Segmenten (das *Basissegment*) und den Wert $\log_2(i)$ bestimmt hat, ist die Berechnung der Adresse des N. By-

te im *Huge*-Segment einfach. Mit dem höherwertigen Wort des Wertes *N* (das ist *N*/64 Kbyte), der um den Wert $\log_2(i)$ nach links verschoben wird (um den *DosGetHugeShift*-Wert) und dem Addieren des Basissegmentselektors (wurde von *DosAlloc-Huge* zurückgegeben) erhält man die Adresse des N. Byte. Der resultierende Wert ist der Segmentselektor des Komponentensegments, die niederwertigen 16 Bits von *i* stellen den Offset im Segment dar. Die Berechnung erfordert nicht viel Zeit, da sie nur eine Verschiebung und eine Addition umfaßt.

Huge-Segmente lassen sich mit einem *DosReallocHuge*-Aufruf vergrößern oder verkleinern. Beim Vergrößern eines *Huge*-Segmentes kann das Anlegen von weiteren physikalischen Komponentensegmenten nötig werden. Da die Regeln zur Adreßgenerierung vorschreiben, welchen Selektor dieses neue Segment haben darf, ist die Vergrößerung eines *Huge*-Segments nicht möglich, wenn der Selektor bereits anderweitig belegt wurde. *DosAllocHuge* übernimmt einen Parameter, der die maximale Größe des *Huge*-Segments bestimmt. Anhand des Wertes werden genügend Selektoren zur Segmenterweiterung freigehalten. Anwendungsprogramme sollten keinen unrealistisch hohen Wert für dieses Argument übergeben, da andernfalls LDT-Selektoren unbenutzt bleiben.

Aus dem Gesagten ergibt sich, daß die vom 8086-Prozessor bekannte Segmentarithmetik mit OS/2 nicht völlig aufgegeben wurde, sondern in veränderter Form durch die Methode der Belegung von *Huge*-Segmenten fortgeführt wird. Anwendungsprogramme für den 8086 verwenden die Segmentarithmetik zur Adressierung von Speicherbereichen mit mehr als 64 Kbyte Umfang. Sie benutzen dafür einen Verschiebewert von 12. Wenn der Wert 2^{12} zum Segmentregisterwert des 80286 addiert wird, zeigt das Segmentregister auf eine Adresse, die um den Wert $2^{12}*16$ (oder 64 Kbyte) im physikalischen Speicher verschoben ist. Der Offsetwert zwischen den Komponentensegmenten ist jeweils 4096, was mit der Adressengenerierung des 8086 zusammenhängt. Obgleich die Berechnung der Segmentwerte im *Protect*-Modus ähnlich ist, sind doch einige Unterschiede zu erkennen. Bei der Berechnung im *Protect*-Modus hat der Seg-

mentselektorwert keine unmittelbare Beziehung zu anderen Selektoren, die das *Huge*-Objekt ausmachen. Der Trick funktioniert nur, weil OS/2 gleichweit auseinanderliegende Selektoren gewählt hat und jeder auf einen physikalischen Speicherbereich der entsprechenden Größe zeigt. In Bild 9.2 sind die Ähnlichkeiten und Unterschiede zwischen der Adressierung des *Huge*-Modells im *Real*- und *Protect*-Modus gezeigt. Der Code im Anwendungsprogramm ist identisch: Ein Segmentselektor wird durch die Addition von $N * i$ zum Basisselektor berechnet. Im *Real*-Modus besitzt i immer den Wert 4096, im *Protect*-Modus wird i von OS/2 bereitgestellt.

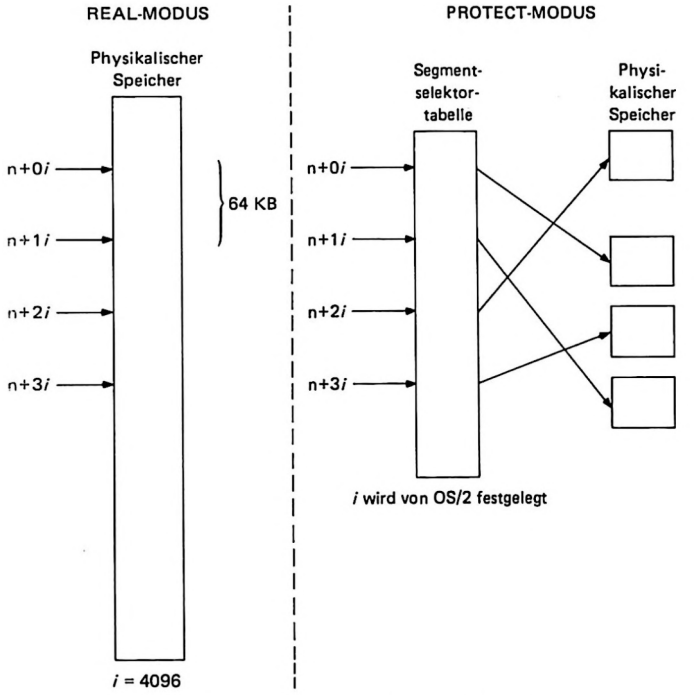


Bild 9.2 Huge-Modell-Adressierung (Real- und Protect-Modus)

Durch die Ähnlichkeiten der Segmentarithmetik des 8086 und der Adressierung von *Huge*-Segmenten unter OS/2 ist die Entwicklung von Programmen, die in beiden Modi funktionieren, einfach. Durch die Verwendung des Verschiebewertes 12 im *Real*-Modus und die Benutzung des von OS/2 bereitgestellten Wertes im *Protect*-Modus kann dieselbe Funktion ohne Änderungen in beiden Modi ablaufen.

9.2.3 Ausführung aus Datensegmenten

Es wurde gezeigt, wie OS/2 durch die Bereitstellung von *Huge*-Segmenten die Beschränkung der Hardware bezüglich der Segmentgröße umgeht. Eine ähnliche Funktion erfüllt OS/2 bei einer weiteren Hardwarebeschränkung, nämlich der Tatsache, daß von Datensegmenten aus keine Programme ausgeführt werden können. Dieses Leistungsmerkmal wird benötigt, obwohl das Laden, Abschneiden und Auslagern von Codesegmenten möglich ist. Einige ablaufoptimierte Programme - beispielsweise der *Presentation Manager* - kompilieren Code zur Ausführung zeitkritischer Tasks "auf der Stelle" (Verschieben von Bits im EGA-Anzeigemodus). Die optimale Befehlsfolge hängt dabei von mehreren Faktoren ab, so daß ein Programm den Code kompilieren und ausführen muß, um eine deutlich spürbare Effizienzsteigerung bei der Ausführung einer Aufgabe zu erlangen. OS/2 unterstützt dieses Leistungsmerkmal über den *DosCreateCSAlias*-Aufruf.

Ein *DosCreateCSAlias*-Aufruf wird mit dem Selektor eines Datensegmentes aufgerufen und legt einen davon unabhängigen Codesegmentselektor (für die 80286-Hardware) an, der aber auf die gleiche Speicherstelle wie der Datensegmentselektor zeigt. Das Resultat ist, daß das Programm nicht aus dem eigentlichen Datensegment, sondern aus einem Codesegment heraus ausgeführt wird. Da Code- und Datensegment in diesem Fall exakt übereinstimmen, wird der gewünschte Effekt erreicht. Der Programmierer muß nur darauf achten, daß der Datenselektor zum Schreiben in das Segment und der Codeselektor zur Ausführung des Codes verwendet wird.

9.2.4 Speicherunterbelegung

Alle bisher angesprochenen Speicherobjekte waren Segmente. OS/2 stellt ein weiteres Leistungsmerkmal bereit: die *Speicherunterbelegung*. Hierbei werden Teile des Speicherbereichs eines Anwendungssegmentes belegt. Die Speicherbereiche können in einem Segment vergrößert, verkleinert und freigegeben werden. OS/2 verwendet einen klassischen *Heap*-Algorithmus, um diese Aufgabe ausführen zu können. Der *DosSubAlloc*-Aufruf verwendet Bereiche, die durch frühere *DosSubFree*-Aufrufen verfügbar gemacht wurden (sofern möglich). Das Segment wird vergrößert, wenn der freie *Heap*-Bereich zu klein wird. Die von einem *DosSubAlloc*-Aufruf zurückgegebenen Speicherbereiche werden *Heap*-Objekte genannt.

Das Speicherunterbelegungspaket arbeitet innerhalb eines Prozesses. Es wird kein Speicherbereich aus dem Systemvorrat verwendet, sondern ein bereits dem Prozeß zugeordneter Speicherbereich. Mit dem Unterbelegungspaket werden nicht einmal Segmente belegt, es dient lediglich der Zerlegung der verfügbaren Segmente des aufrufenden Programms (das aufrufende Programm muß mindestens eine Zugriffsberechtigung darauf besitzen). Die Neubelegung von Segmenten wurde ausgeschlossen, da der Speicherschutz auf Segmentbasis besteht. Falls das Unterbelegungspaket ein dem Gesamtsystem zur Verfügung stehendes Segment beanspruchen würde, könnten Bereiche anderer Prozesses durch die Belegung für das Heap-Objekt zerstört werden. In Bild 9.3 ist die Funktionsweise der Speicherunterbelegung dargestellt. Das Segment *j* wird unterteilt. *H* ist der Anfangsteil (*header*) der Unterbelegung, die Schattierungen kennzeichnen freie Speicherbereiche.

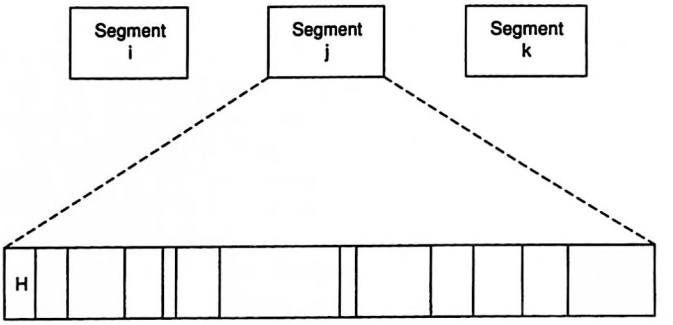


Bild 9.3 Speicherunterbelegung

Die Speicherunterbelegung unterteilt Segmente, auf die ein Klientenprozeß zugreifen kann. Das bedeutet, daß auch gemeinsam benutzte Speichersegmente unterteilt werden können. Diese Technik dient der Kommunikation zwischen zwei oder mehreren Prozessen, die ein gemeinsam verwendetes Speichersegment besitzen. Das Risiko bei dieser Methode ist, daß ein Prozeß die *Heap*-Objekte eines anderen Prozesses zerstören kann.

Im Zusammenhang mit den Ausführungen über dynamische Verknüpfungssubsysteme wurden die Programmiertechniken zur Entwicklung eines verläßlichen Subsystems beschrieben. Das OS/2-Speicherunterbelegungspaket ist ein gutes Beispiel für ein solches Subsystem. Daher soll nun näher darauf eingegangen werden. Der erste Versuch zur Entwicklung des Unterbelegungspaketes führte zu einem geradlinigen *Heap*-Belegungsmechanismus (ähnlich dem in C- oder Pascal-Laufzeitbibliotheken enthaltenen). Es gab eine Kette von *Heap*-Objekten, die auf Anforderung hin einem Prozeß zugeordnet wurden. War das kleinste Heap-Objekt immer noch größer als der Umfang der Speicheranforderung, wurde es in einen belegten und einen freien Teil gesplittet. Freigegebene Heap-Objekte wurden mit zusammenhängenden freien Objekten verknüpft. Das Unterbelegungspaket arbeitete mit einem Segmentzeiger und weiteren Argumenten und lieferte mehrere Rückgabeparameter - einen

Offset und die veränderten Segmentdaten (*Heap-Header*). Wenn der veränderte Status des bereitgestellten Datensegmentes als Rückgabewert interpretiert wird, ähnelte das Unterbelegungspaket einer Funktion: Es besitzt keinen eigenen Status, sondern liefert lediglich Rückgabewerte, die aus den Eingabeargumenten berechnet wurden. Die einfache Dynlink-Unterbelegungsroutine verwendete keine globalen Datensegmente und benötigte nicht einmal temporär ein Datensegment.

Diese einfache Implementierung hat einen gravierenden Nachteil: Es konnte nur immer ein Prozeß die Routine zur Verwaltung gemeinsam verwendeter Speichersegmente einsetzen, die gleichzeitige Benutzung durch mehrere Threads eines Prozesses war ebenfalls nicht möglich. Die Liste des verfügbaren Heap-Speichers ist ein kritischer Bereich; rufen mehrere Threads die Unterbelegung für das gleiche Segment auf, kann dies zu Fehlern in der Liste führen. Aus diesem Grund mußte die Unterbelegung durch die Benutzung einer Semaphore erweitert werden, um den kritischen Bereich zu schützen. Da nur die Verwendung mehrerer Threads eines Prozesses und keine gemeinsamen Speichersegmente unterstützt werden sollen, wurden RAM-Semaphoren implementiert, die sich in dem zu schützenden Bereich selbst befinden konnten. Die Semaphore kann bei der unerwarteten Beendigung eines Prozesses gesetzt bleiben, wenn das betroffene Segment nicht zur gemeinsamen Benutzung durch mehrere Prozesse bestimmt ist. Das Segment wird in jedem Fall zerstört und muß daher nicht explizit freigegeben werden.

Aber selbst bei der recht einfachen Verwaltung lokaler Segmente müssen eine Reihe besonderer Situationen berücksichtigt werden. Ein Problem stellen Signale dar: Was geschieht, wenn Thread 1 die Unterbelegung aufruft und zu diesem Zeitpunkt ein Signal ankommt, beispielsweise SIGINT aufgrund der Betätigung der Tastenkombination Ctrl-C? Thread 1 wird zur Bearbeitung des Signals bei seiner derzeitigen Aufgabe im kritischen Bereich des Unterbelegungspaketes unterbrochen. Häufig führt dies nicht zu einem Fehler, da die Bearbeitung des unterbrochenen Programms nach der Abarbeitung der signalbehandelnden Routine wieder aufgenommen wird. Es kann aber vorkommen,

daß kein Rücksprung, sondern eine Verzweigung zur Befehlseingabeschleife erfolgt oder aus der signalbearbeitenden Routine heraus die Unterbelegung aufgerufen wird. In beiden Fällen kommt es zu einer unlösbaren Situation (*deadlock*), die durch die Verwendung des *DosHoldSignal*-Aufrufes vermieden werden kann. *DosHoldSignal* ist das Äquivalent für Signale zur Anweisung CLI für Hardwareinterrupts: Es wird eine Verzögerung des Signals/Interrupts erreicht. Das Signal bzw. der Interrupt wird zurückgehalten, bis das Anwendungsprogramm Signale oder Interrupts wieder zuläßt. Beachten Sie, daß Signale nicht für mehr als ein bis zwei Sekunden verzögert werden sollten. Vor der Anforderung einer Semaphore auf einen kritischen Bereich sollte eine Signalverzögerung programmiert werden. Ein Schutz vor unlösbaren Situationen ist gegeben, wenn Signale erst nach der Freigabe der Semaphore wieder auftreten dürfen.

Im Gegensatz zur Anweisung CLI können *DosHoldSignal*-Aufrufe geschachtelt werden. OS/2 zählt die Anzahl der ausgeführten *DosHoldSignal*-Sperraufrufe und läßt Signale erst nach der Ausführung derselben Anzahl an Freigabeaufrufen wieder zu. Eine Routine kann folglich ohne Beeinflussung des aufrufenden Programms ein Sperr-/Freigabeaufrufpaar ausführen. Falls das aufrufende Programm vor dem Aufruf des Kindprozesses Signale verzögert hat, bleibt dieser Zustand erhalten. In jedem Fall sorgt der Freigabeaufruf des Kindprozesses dafür, daß der ehemalige Zustand bei der Wiederaufnahme des aufrufenden Programms wiederhergestellt wird.

Sobald ein Dynlink-Paket Veränderungen im Status des aufrufenden Programms vornimmt, muß sichergestellt sein, daß der ursprüngliche Zustand vor dem Rücksprung wiederhergestellt wird. Funktionen, die eine Schachtelung erlauben, tragen automatisch dafür Sorge. Für alle anderen Funktionen muß der ursprüngliche Status explizit gespeichert und nach der Beendigung der Funktion gesetzt werden.

Hiermit sind aber noch nicht alle auftretenden Probleme gelöst. Ein zweites Problem stellt das Leistungsmerkmal *DosExitList* dar. Falls sich ein Klientenprozeß in einem kritischen Bereich

im Unterbelegungspaket befindet und der Klient plötzlich beendet wird - er könnte mit einem *Kill*-Aufruf gestoppt worden sein oder es ist ein GP-Fehler aufgetreten - kann dies dazu führen, daß der Prozeß nicht sofort beendet wird. Sofern *DosExitList*-bearbeitende Routinen vorhanden sind, werden diese ausgeführt. Falls hierbei ein Aufruf des Unterbelegungspaketes auftritt, führt dies wiederum zu einer unlösbaren Situation (*deadlock*). Diese Problemstellung könnte durch ein Verbot des Aufrufs des Unterbelegungspaketes aus einer Exit-Liste heraus gelöst werden. Für einige Dynlink-Subsysteme ist dies sicherlich eine akzeptable Lösung, als Einschränkung von OS/2 aber nicht tragbar. Dieses Problem muß also anderweitig gelöst werden.

Der *DosHoldSignal*-Trick kann nicht eingesetzt werden, da hierdurch zwar externe Beendigungen abgesichert wären, nicht aber GP-Fehler. Man kann dies sicherlich aus dem Blickwinkel sehen, daß beim Auftreten eines GP-Fehler sowieso nichts mehr hilft. Falls aber das Programm oder ein davon verwendetes Dynlink-System *DosExitList*-bearbeitende Routinen besitzt und diese einen Heap-Speicherbereich belegen oder freigeben, bleibt der Prozeß hängen und kann nicht korrekt beendet werden. Um die Anwendung aus dem Speicher zu entfernen, wäre ein erneuter Systemstart nötig. Die Lösung besteht in der Verwendung einer System- statt der RAM-Semaphore zum Schutz des Speichersegments. Systemsemaphoren sind in der Verwendung zwar etwas langsamer als RAM-Semaphoren, bieten dafür aber eine Reihe von Vorteilen. So können Systemsemaphoren zur *exklusiven Verwendung* bestimmt werden, das heißt, daß nur der Thread, der Besitzer ist, die Semaphore freigeben kann. Damit verknüpft ist ein Leistungsmerkmal, das der Feststellung der Beendigung des Besitzers dient. Die *DosExitList*-behandelnde Routine kann auf diese Weise feststellen, ob eine Semaphore verwaist ist (siehe Kapitel 16), und das Unterbelegungspaket vermag seine kritischen Bereiche durch die Verwendung einer Exklusivsemaphore zu schützen. Hinzu kommt die Festlegung einer *DosExitList*-behandelnden Routine, die die verwaiste Semaphore freigibt. Der *Exitlist*-Code kann erfahren, ob ein Thread seines Prozesses eine Semaphore verwaist zurückgelassen

hat und die Semaphore in einem solchen Fall freigeben. Die Freigabe der Semaphore allein nützt allerdings nichts, wenn die Heap-Header in einem inkonsistenten Zustand vorliegen. Das Unterbelegungspaket kann entweder so entwickelt werden, daß die Header immer in einem konsistenten Zustand vorliegen oder aber so, daß der Status der modifizierten Header zur Wiederherstellung der Heap-Struktur gespeichert wird.

Im letztgenannten Fall sollten Sie sicherstellen, daß die von Ihnen erstellte *DosExitList*-bearbeitende Routine als erste aufgerufen wird (siehe *DosExitList*-Dokumentation).

Auch wenn die Klientenanwendung bei der Abarbeitung der *DosExitList*-bearbeitenden Routine keine Unterbelegungsaufrufe ausführen darf, soll die Speicherunterbelegung doch die Belegung eines gemeinsam verwendeten Speichersegmentes erlauben. Aus diesem Grund verwendet das OS/2-Unterbelegungspaket *DosExitList*, um die Unterbelegungsstrukturen und Semaphoren freigegeben zu können, falls ein Klienten-Thread bei der Abarbeitung eines kritischen Bereiches des Unterbelegungspaketes beendet wird.

Das Dynlink-Unterbelegungspaket gibt nicht nur Einblicke in die Konzeption von Subsystemen, sondern zeigt darüber hinaus den Wert einer Systemarchitektur, die Dynlink-Routinen als Standardschnittstellen einsetzt (unabhängig von der Art des Programms, das den Service bereitstellt). Das Speicherunterbelegungspaket ist in OS/2 Version 1.0 nicht Bestandteil des Kernels, sondern ein externes Unterroutinensystem. In der OS/2-Version für die 80286-Umgebung wird dieser Ansatz fortgeführt werden. Bei einem 80386-OS/2 wird es hingegen zu einer anderen Vorgehensweise kommen. Die 80386-Architektur unterstützt die seitenorientierte Verwaltung virtuellen Speichers, so daß das Speicherseitenmanagement (*Paging*) als Teil des Segments stattfinden kann. Dadurch mag es bei OS/2 zu Änderungen in der Speicherunterbelegung kommen. So wäre eine Änderung der Heap-Struktur im Sinne eines optimierten Paging denkbar oder der Kernel wird in den Vorgang einbezogen, um das Paging zu unterstützen. In jedem Fall wird es in künftigen

OS/2-Versionen problemlos sein, das externe Speicherunterbelegungssystem weiter auszubauen, da die interne Kompatibilität durch die Dynlink-Routinen als Standardschnittstelle gewährleistet ist.

9.3 Speicherauslagerung

Eines der wichtigsten Merkmale der Speicherverwaltung des 80286 ist die *Speicherauslagerung* (*swapping*). Bei dieser Technik werden Code- oder Datensegmente auf einen Datenträger ausgelagert und dadurch Hauptspeicherkapazität zur Wiederverwendung freigegeben. Die ausgelagerten Speicherbereiche werden bei Bedarf wieder in den Hauptspeicher geladen. Auf diese Weise ist die gleichzeitige Ausführung von Programmen möglich, für die die tatsächlich vorhandene Speicherkapazität nicht ausreichen würde. Da nicht benötigte Programme auf einen Datenträger ausgelagert werden, muß nur genügend Hauptspeicher zur Aufnahme der zu einem bestimmten Zeitpunkt ausgeführten Programme vorhanden sein. Bei der erneuten Aktivierung eines ausgelagerten Programms werden die benötigten Speicherbereiche geladen und im Gegenzug gegebenenfalls andere Bereiche ausgelagert.

Die Auslagerung der Programme findet auf Segmentbasis statt. Dabei spielt die Zusammengehörigkeit von Segmenten keine Rolle, das heißt, die Segmente werden nicht zu Gruppen zusammengefaßt. Die Auslagerung nur eines Teils eines Segmentes erlaubt der 80286 nicht, es muß sich stets um vollständige Segmente handeln. Die obige Beschreibung gibt die tatsächlichen Verhältnisse vereinfacht wieder, um das Prinzip zu verdeutlichen. Es können nicht nur vollständige Prozesse ausgelagert werden, sondern auch ein oder mehrere Segmente. Die restlichen Segmente verbleiben unverändert im Hauptspeicher. Solange ein Prozeß keine ausgelagerten Segmente benötigt, kann dieser ungehindert ablaufen. Beim Zugriff auf ein ausgelagertes Segment erzeugt die 80286-Hardware ein besonderes Signal (*segment fault trap*), das OS/2 abfragt. Die fehlerbearbeitende Routine für das Segment lädt das benötigte Segment (gegebe-

nenfalls wird zuvor ein anderes Segment ausgelagert) und der Prozeß läuft weiter ab. Segmentfehler sind für einen Prozeß nicht erkennbar, er läuft mit der Ausnahme der zur Ausführung der Lade-Anweisung benötigten Zeit (30 Millisekunden statt drei Mikrosekunden) wie gewöhnlich ab.

Bei der Anforderung von erneut zu ladenden Bereichen muß gegebenenfalls ein Segment ausgelagert werden. Hierbei ist es wichtig, daß das richtige Segment ausgewählt wird. Stellen Sie sich einen Prozeß vor, der abwechselnd eine Referenz auf die Segmente A und B ausführt. Wird zur Ausführung einer Referenz auf Segment A das Segment B ausgelagert, führt dies nach der Abarbeitung von nur wenigen Anweisungen dazu, daß Segment B wieder geladen wird, wenn die nächste Referenz darauf verweist. Eine Auslagerung des Segmentes A bewirkt, daß das Segment bald wieder geladen werden muß und sich dieser Prozeß wiederholt. Der Durchsatz des Systems sinkt drastisch. Der Vorgang des gegenseitigen Austauschens aktiver Segmente benötigt 10.000 mal mehr Zeit als dies der Fall wäre, wenn beide Segmente im Hauptspeicher gehalten werden. Dies erscheint auf den ersten Blick viel, ist jedoch tragbar, *sofern die Auslagerung nicht ständig erfolgt.*

Es ist also ersichtlich, daß die Leistungsfähigkeit des Systems wesentlich von der Auswahl des auszulagernden Segmentes abhängt. OS/2 verwendet das LRU-Prinzip (*least recently used*), bei dem das Segment ausgelagert wird, das am längsten nicht mehr benutzt wurde. Idealerweise sollte man das Segment wählen, das vom Zeitpunkt der Entscheidung erst nach der längsten Zeitspanne wieder verwendet wird. Leider kann das Betriebssystem aber das Verhalten von Prozessen nicht vorhersehen und muß daher von abgelaufenen Vorgängen ausgehen. Die Annahme, daß das am längsten nicht genutzte Segment auch in naher Zukunft nicht genutzt werden wird, ist sicherlich vernünftig, wenngleich nicht immer zutreffend.

Bei der LRU-Methode ist die nachfolgend beschriebene Situation denkbar, die die Systemleistung drastisch reduziert. Falls ein Prozeß *N* Segmente nacheinander anspricht und anschlie-

ßend wieder beim ersten beginnt, im Hauptspeicher aber nur *N-1* Segmente vorliegen können, führt die Auslagerung eines Segmentes nach der LRU-Methode dazu, daß jeweils das nächste benötigte Segment ausgelagert wird. Eine Ausweitung dieser Thematik ist an dieser Stelle nicht möglich. Programmierer, die in ihren Programmen wiederholt Code- oder Datenpassagen ansprechen müssen, sollten sich weitere Literatur zur Verwendung virtueller Speichersysteme besorgen. Beachten Sie, daß OS/2 auf einem 80286 nur auf Segmentbasis auslagert, auf dem 80386 wird die Auslagerung seitenorientiert erfolgen (in Seiten zu je vier Kbyte).

Der Auslagerungsalgorithmus folgt dem LRU-Prinzip ohne Berücksichtigung von Thread- und Prozeßprioritäten. Als auslagerbar gekennzeichnete Systemsegmenten wird keine besondere Behandlung zuteil. Einige Systemsegmente sind als nicht auslagerbar markiert. Hierzu gehören beispielsweise das Segment mit dem Programmcode zur Auslagerungsverwaltung und der Treiber für den Zugriff auf den Datenträger. Ein Teil des Kernel- und Einheitentreibercodes ist ebenfalls nicht ausgelagerbar, da diese Programmteile zur Bearbeitung eines Interrupts benötigt werden. Das Nachladen würde zuviel Zeit kosten und es könnten Interferenzen zwischen dem ausgelagerten interruptbearbeitenden Code und dem interruptbearbeitenden Teil des Einheitentreibers, der in die Auslagerung einbezogen ist, auftreten. Teile des Kernel-Codes werden auch im *Real*-Modus bei einer Anforderung der DOS-3X-Box benötigt. *Real*-Modus-Programmteile können nicht ausgelagert werden, da der Prozessor die Segmentauslagerung in diesem Modus nicht unterstützt.

Die Methode, mehr Programme ablaufen zu lassen, als es der Hauptspeicher eigentlich zuläßt, wird *Verwendung des virtuellen Speichers* (*memory overcommit*) genannt. OS/2 muß auf die Menge des ausgelagerten Speichers achten, um nicht in eine Situation zu geraten, in der das erneute Laden eines Segments trotz Auslagerung nicht mehr möglich ist. Dies hätte nämlich zwei Folgen: Zum ersten könnte der Benutzer die (vielleicht seit Stunden) bearbeiteten Daten nicht sichern und zum zweiten gäbe es keine Möglichkeit, dem Programm mitzuteilen, was pas-

siert ist. Das Programm könnte einfach nicht mehr ausgeführt werden. Aus diesem Grund verfolgt OS/2 die Belegung des virtuellen Speichers auf zwei Weisen. Vor dem Start wird zum einen sichergestellt, daß genügend Auslagerungskapazität für den zu ladenden Prozeß vorhanden ist. Zum anderen sorgt OS/2 dafür, daß zu jedem Zeitpunkt genügend RAM-Kapazität frei ist, um einen Auslagerungsvorgang ausführen zu können.

Auf den ersten Blick scheint es einfach zu sein, festzustellen, ob die RAM-Kapazität ausreicht - entweder kann das Programm im Hauptspeicher ausgeführt werden oder nicht. Unter OS/2 ist dies jedoch etwas schwieriger, da Segmente eines Programms oder einer Dynlink-Routine erst bei Bedarf geladen werden können. Beim Start eines Programms kann dieses theoretisch (unter Verwendung aller gegebenenfalls benötigten Module) nahezu unendlich viel Speicherkapazität für sich beanspruchen. Bei der Speicheranforderung durch ein Programm kann OS/2 die Speicherzuweisung verweigern und einen Fehlercode zurückgeben, wenn nicht genügend Kapazität bereitsteht. Das Programm muß dann Abhilfe schaffen (Anforderung eines kleineren Speicherbereiches, Ignorieren der Benutzereingabe, Ausgabe einer Fehlermeldung oder ähnliches).

OS/2 muß keine besondere Behandlung der expliziten Speicheranforderungen vorsehen, da solche Anforderungen immer abgelehnt werden können. Hier sind die impliziten Speicheranforderungen das Problem, beispielsweise bei nachzuladenden Segmenten. Es gibt nicht nur keine Schnittstelle zur Übergabe eines Fehlercodes (zum Laden eines Segmentes wird eine *Lade-Segmentregister*-Anweisung verwendet, die keinen Fehlercode liefert), sondern es kann vorkommen, daß das Programm nicht weiter ausgeführt wird, wenn das angeforderte Programmteil nicht geladen werden kann. OS/2 summiert vor dem Laden eines Programms (über *DosExecPgm*) die Größe aller Segmente, die keine Pure-Segmente sind (auch wenn diese als "ladbar auf Anforderung" markiert sind). Dieselbe Berechnung wird für die in den Ladevorgang einbezogenen Dynlink-Bibliotheken und deren Referenzen usw. ausgeführt. Diese Summe plus dem internen System-Overhead stellt die maximale implizite Speicher-

anforderung des Programms dar. Sofern diese Auslagerungsspeicherkapazität verfügbar ist, wird das Programm geladen.

Die Ausführung eines Programms erfolgt letztlich im RAM. Daher muß OS/2 nicht nur die benötigte Auslagerungskapazität berücksichtigen, sondern auch die erforderliche RAM-Kapazität. Das Kriterium ist dabei nicht der tatsächlich freie RAM-Speicher beim Laden des Programms, sondern der durch Auslagerung freizustellende Platz. Falls im RAM ein auslagerbares Segment vorliegt, kann dieses auch ausgelagert werden, da immer genügend Kapazität zur Auslagerung freigehalten wird. Segmente, die aus einer Datei geladen werden können, werden üblicherweise nicht ausgelagert, sondern einfach gelöscht. Bei einem erneuten Zugriff lädt OS/2 die Segmente erneut aus der .EXE- oder .DLL-Datei.

Hinweis: Eine Ausnahme hiervon bilden Programme, die von austauschbaren Datenträgern aus ausgeführt werden. OS/2 lädt alle benötigten Segmente von der .EXE- oder .DLL-Datei und lagert sie bei Bedarf aus. Dadurch werden Hardwarefehlern bei unlösbaren Situationen (deadlock) und Fehlern in der Datenträgerverwaltung vorgebeugt.

Da sich bei der Ausführung eines Prozesses nicht alle Segmente im Hauptspeicher befinden müssen, wird lediglich genügend Kapazität zur Aufnahme von sechs 64-Kbyte-Segmenten benötigt. Dieser Wert ergibt sich aus der Architektur des 80286, es ist die maximale Speicherkapazität, die zur Ausführung eines Prozesses benötigt wird. Schon zur Ausführung einer einzelnen Anweisung müssen sich die vier durch die Segmentregister bezeichneten Segmente im Hauptspeicher befinden. Die anderen beiden Segmente ergeben sich aus dem *Worst-Case*-Fall einer *far return*-Anweisung, die aus einem Ring-2-Segment heraus ausgeführt wird (siehe 18.1). Zum Start der Anweisungsausführung müssen die in den vier Segmentregistern angegebenen Segmente verfügbar sein, zu deren Beendigung müssen die beiden neuen Segmente (CS und SS), auf die sich die *far return*-Anweisung bezieht, ebenfalls vorliegen. Aus dieser Überlegung ergibt sich die maximale Anzahl gleichzeitig verfügbarer Segmente zu 6 Segmenten. Der Wert 64 Kbyte resultiert aus dem Maximalumfang eines Segments. Solange OS/2 die sechs Segmente

auslagern und verändern kann, ist jedes ausgelagerte Programm ausführbar.

Es ist klar, daß das Antwortverhalten eines Systems mit minimaler Speicherausstattung zu wünschen übrig läßt. Üblicherweise ist auch mehr Speicherkapazität verfügbar. Die Verwaltung des virtuellen Speicherplatzes muß nur sicherstellen, daß alle Programme ablaufen können, das Laden eines Prozesses wird nicht unterbunden, wenn dadurch die Antwortzeit des Systems verschlechtert wird. Es kann vorkommen, daß ein Benutzer weniger Hauptspeicher einbaut, weil er relativ lange Antwortzeiten als akzeptabel empfindet. Vielleicht kommen merkliche Verzögerungen nur selten vor und der Benutzer ist Willens, diese ab und zu hinzunehmen.

Allgemein gilt: Wenn ein System zu häufig Segmente auslagert, ist das kein schwerwiegender Fehler! Der Benutzer weiß, wo das Problem liegt, er weiß, wie das Problem zu beheben ist (mehr Hauptspeicher einbauen oder weniger Programme zu einem Zeitpunkt ablaufen lassen) und kann zwischenzeitlich weiterarbeiten.

Da alle Segmente einer Anwendung auslagerbar sind, hat der Start einer Anwendung keinen Einfluß auf die verfügbare Hauptspeicherkapazität (es ist genügend Auslagerungskapazität vorhanden). Zu einem Engpaß beim Hauptspeicher kann es aber durch das Laden nicht auslagerbarer Segmente durch Einheitentreiber kommen. Daher kann das Laden oder eine Speicherbelegungsanforderung eines Einheitentreibers abgelehnt werden, wenn dies zu weniger als sechs frei verfügbaren 64-Kbyte-Segmenten führen würde.

9.3.1 Hinweise zur Auslagerung

Die Auslagerungskapazität des Systems liegt in der Datei SWAPPER.DAT, die beim Systemstart angelegt wird. Die Position der Auslagerungsdatei wird in der Datei CONFIG.SYS festgelegt. OS/2 muß nicht die benötigte Gesamtkapazität der Datei beim Start belegen, statt dessen kann die Datei bei Bedarf bis zu ihrer maximalen Größe ausgedehnt werden. Unter OS/2 Version

1.0 kann die Auslagerungsdatei zwar anwachsen, aber nicht mehr verkleinert werden.

Die verfügbare Auslagerungskapazität unter OS/2 setzt sich aus der Auslagerungsdatei und Extra-RAM zusammen. Sicherlich kann ein System mit acht Mbyte RAM und einer maximalen Größe der Auslagerungsdatei von 200 Kbyte auch Programme ausführen, deren Speicherkapazitätsbedarf die 200-Kbyte-Grenze übersteigt. Nach der Abgrenzung der nicht auslagerbaren sowie der sechs benötigten 64-Kbyte-Segmente wird der restliche Hauptspeicher für Auslagerungszwecke vorgesehen.

Es wurde bereits kurz angeschnitten, daß im *Real*-Modus verwendete Speicherbereiche nicht ausgelagert werden können. Das bedeutet, daß die gesamte DOS-3X-Box nicht auslagerbar ist. Tatsächlich ist OS/2 aufgrund der Speicheranforderungsmethoden von MS-DOS gezwungen, zwischen Speicherbereichen im *Real-* und *Protect*-Modus eine starre Grenze zu ziehen. Der Speicherbereich unterhalb der RMSIZE-Grenze, der in der Datei CONFIG.SYS ausgewiesen ist, wird ausschließlich *Real*-Modus-Programmen zugeordnet. Ausgenommen davon sind die für Einheitentreiber und Teile des OS/2-Kernels benötigten Bereiche, die im *Real*-Modus ablaufen.

In einer frühen Entwicklungsstufe von OS/2 wurde erwogen, Protect-Modus-Segmente in unbenutztem Real-Speicher abzulegen. Der Gedanke wurde aus zwei Gründen bald wieder fallengelassen. Zum einen besteht die Gefahr, daß das Real-Modus-Programm das Segment zumindest in Teilen überschreibt. Genau genommen handelt es sich dabei zwar um einen Fehler im Real-Modus-Anwendungsprogramm, unter MS-DOS bleibt dieser Fehler jedoch ohne Auswirkungen. Es läßt sich daher vermuten, daß viele MS-DOS-Programme diesbezüglich Fehler aufweisen, die bislang noch nicht aufgefallen sind, unter OS/2 jedoch zum Vorschein kämen.

Ein zweiter Grund hängt mit Real-Modus-Anwendungen zusammen, die für eine Singletasking-Umgebung entwickelt wurden. Eine solche Anwendung fordert üblicherweise ein Mbyte Speicherkapazität an, was - wie bekannt - vom System abge-

lehnt werden muß. Die Ablehnung wird der Anwendung mit einem Fehlercode mitgeteilt. Mit der Ablehnung erhält das Programm zugleich Informationen über die zum Zeitpunkt des Systemaufrufs frei verfügbare Speicherkapazität. Eine Real-Modus-Anwendung fordert dann im allgemeinen genau diese Speicherkapazität an, prüft dabei jedoch nicht mehr auf Fehlercodes, da es in einem Singletasking-System nicht vorkommt, daß Kapazität zwischenzeitlich anderweitig verwendet wurde. In einer Multitasking-Umgebung führt ein solches Verhalten zu Problemen, da zwischen der ersten und der zweiten Speicheranforderung der freie Speicherbereich oder ein Teil davon von anderen Programmen verwendet worden sein kann. Dies ist ein weiterer Grund, weshalb OS/2 den unteren Speicherbereich nur für Real-Modus-Anwendungen einsetzt und zu anderen Zwecken nicht anspricht, auch wenn dieser Speicherbereich dadurch den größten Teil der Zeit nicht genutzt wird.

Wie bereits dargelegt, besteht die primäre Aufgabe von OS/2 darin, eine Umgebung bereitzustellen, in der Programme ablaufen können. Um den Durchsatz der Programmabarbeitung kümmert sich OS/2 nicht, dies ist Aufgabe des Benutzers. Das ist durchaus gerechtfertigt, da es sich bei OS/2 um ein Einbenutzerbetriebssystem handelt. In Mehrbenutzerumgebungen muß das Betriebssystem für einen guten Durchsatz sorgen, da ein Benutzer, der einen Durchsatzverlust verursacht, nicht unbedingt auch der Benutzer ist, der unter diesem Mangel leidet. Auf einem Einbenutzersystem ist der Benutzer selbst für die Systembelastung verantwortlich. Ist die Ablaufgeschwindigkeit nicht zufriedenstellend, muß er die Anzahl der auszuführenden Anwendungen vermindern oder mehr Hauptspeicherkapazität in das System einbauen. Trotzdem sollten Anwendungsentwickler beachten, daß Anwendungen mit umfangreichen Speicheranforderungen die Auslagerungsmöglichkeit von OS/2 nicht über Gebühr beanspruchen.

Alle Probleme im Zusammenhang mit der Optimierung der Auslagerung lassen sich auf einen Punkt bringen: die *Lokalität der Referenz*. Das bedeutet, daß zusammengehörige Speicherstellen sowohl räumlich als auch zeitlich nahe beieinander ge-

halten werden. Bei der Verfügbarkeit von fünf Funktionen in einem Programm sollten diese im gleichen Segment vorliegen, damit andere, nicht benötigte Segmente ausgelagert werden können (was nicht der Fall ist, wenn Teile der Funktionen im auszulagernden Segment vorliegen).

Ein guter Teil der Literatur beschäftigt sich mit dem Verhalten virtueller Speicherverwaltungssysteme in Großrechnerumgebungen. Die darin wiedergegebenen Überlegungen wurden zu einer Zeit angestellt, als Hauptspeicher noch sehr teuer war. Welche Segmente resident und welche auslagerbar sein sollten, konnte nur mit großem Aufwand bestimmt werden. Da Speicherbausteine teuer und Auslagerungseinheiten relativ schnell waren, wurden Programme damals so entwickelt, daß möglichst viel freier Speicher durch weitgehende Auslagerung geschaffen wurde. Falls ein in naher Zukunft benötigtes Segment ausgelagert wurde, konnte es schnell wieder geladen und weitergearbeitet werden. Heute stellt sich die Situation umgekehrt dar. Speicherkapazität ist preiswert zu haben und die Einheiten (im allgemeinen Festplatten), auf denen die Auslagerungen vorgenommen werden, sind im Vergleich dazu langsam. Die in OS/2 implementierte Strategie sieht konsequenterweise die Auslagerung von Segmenten vor, die auf absehbare Zeit vermutlich nicht benötigt werden, sie ist aber nicht darauf ausgerichtet, Segmente kurzfristig auszulagern.

Eine Ableitung des Konzepts der virtuellen Speicherung stellen *Working Sets* dar. Man versteht darunter logisch zusammenhängende Programmteile. Das Working Set für einen Threat umfaßt die Segmente, die "in naher Zukunft" (in einigen Sekunden oder Minuten) benötigt werden. Man kann Programme auf die Zusammenstellung von Working Sets hin untersuchen. Es ist klar, daß sich die Working Sets in Abhängigkeit von der jeweils konkreten Aufgabe, die das Programm durchführt, ändern. Code und Daten sollten auf zwei Segmente verteilt werden, damit die Größe jedes Working Set minimiert wird. So beinhalten Programme häufig umfangreiche Programmteile zur Bearbeitung fehlerhafter Situationen. Diese sollten zu einem Segment zusammengefaßt werden, das nur im Bedarfsfall Hauptspeicherkapazi-

tät benötigt. Im System sollten auch nicht zu viele Segmente auftreten. Häufig zusammen verwendete Funktionen sollten in einem Segment vorliegen; unzusammenhängende Programmteile und Daten sollten hingegen nicht in einem Segment zusammengefaßt werden. Bei der Zusammenstellung ist auch die Größe der Segmente zu berücksichtigen. Zu viele kleine Segmente erhöhen den Verwaltungsaufwand des Systems, während zu große Segmente die Auslagerung verlangsamen (da immer vollständige Segmente ausgelagert und geladen werden). Die Aufsplittung in zwei Segmente macht keinen Sinn, wenn die darin enthaltenen Informationen logisch zusammenhängen, wohl aber, wenn dies nicht der Fall ist.

Eine ausführliche Erläuterung dieser Problemstellung kann im vorliegenden Buch nicht geboten werden. Es empfiehlt sich, vor der Entwicklung speicherintensiver Anwendungssoftware die einschlägige Literatur zu beziehen und die Programme entsprechend auszurichten. In jedem Fall sollte man den Speicherbedarf der Programme so weit wie möglich reduzieren, da dies immer die beste Versicherung gegen Durchsatzverluste bei stark ausgelasteten Systemen ist.

9.4 Status und Information

OS/2 nutzt die LDT- und GDT-Architektur des 80286, die zwei Segmente (*infosegs*) mit Systeminformationen bereitstellt. OS/2 aktualisiert den Inhalt der beiden Segmente (eines ist global, das andere lokal) beim Auftreten einer Änderung. Das globale Informationssegment enthält Informationen über das Gesamtsystem, das lokale Informationssegment enthält prozeßspezifische Daten. Beide Segmente sind schreibgeschützt und haben einen Nur-Lese-Zugriff, wobei alle Prozesse auf das globale Informationssegment zugreifen können, während jeder Prozeß ein eigenes, lokales Informationssegment besitzt.

Das globale Informationssegment enthält Datums- und Zeitangaben. Das Feld "seit 1970 vergangene Sekunden" ist besonders interessant, da es beispielsweise für Zeitmessungen eingesetzt werden kann. Die Berechnung der Zeitspanne zwischen zwei

Zeitangaben ist einfach: Es werden lediglich die beiden Zeitwerte voneinander subtrahiert und das Ergebnis durch die Anzahl der Sekunden, die der gewünschten Zeiteinheit entsprechen (beispielsweise 60 für Minutenangaben), geteilt. Dabei ist zu beachten, daß das Datums-/Zeitfeld aus 32 Bits besteht, der 80286 aber bei einem Lesevorgang nur 16 Bits übernimmt. Beim Lesen eines Datums-/Zeiteintrags zum Zeitpunkt der Aktualisierung kann es vorkommen, daß ein falscher Wert übernommen wird. Der Wert ist fatalerweise nicht um 1 verkehrt, sondern um 63.335. Aus diesem Grund sollte ein gerade gelesener Wert mit der Angabe im globalen Informationssegment verglichen werden. Bei identischen Werte war der Lesevorgang erfolgreich, ansonsten muß der Vorgang *Lesen/Vergleichen* so lange durchgeführt werden, bis die Werte übereinstimmen. Die RAS-Information (RAS steht für *Reliability*, *Availability* und *Serviceability*) wird zu Systemdiagnosezwecken eingesetzt und ist für Programmierer kaum von Interesse.

Das lokale Informationssegment enthält Prozeß- und Thread-Informationen über den derzeit ausgeführten Thread. Der Unterbildschirmgruppenwert (*subscreen group value*) wird vom Subsystem des Presentation Manager eingesetzt und hat für Anwendungsprogramme keine Bedeutung. Weitere Details zu den Informationssegmenten finden Sie im OS/2-Handbuch.

Kapitel 10

Umgebungsstrings

Eine schwerwiegende architektonische Schwäche von MS-DOS liegt in der fehlenden Unterstützung logischer Einheiten- und Verzeichnisnamen. Falls ein Programm beispielsweise eine temporäre Datei auf den schnellsten Datenträger des Systems schreiben will, muß zuvor bekannt sein, welches Laufwerk dazu angesprochen werden soll. Häufig wird hierfür Laufwerk C genutzt, in der Annahme, daß es sich hierbei um die Festplatte handelt. Einige Systeme besitzen aber keine Festplatte, in diesem Fall muß Laufwerk B (ein Diskettenlaufwerk) eingesetzt werden. Es gibt aber auch Systeme, die nur ein Laufwerk (Laufwerk A) besitzen. Und selbst wenn eine Festplatte mit Namen C existiert, kann eine weitere Festplatte unter der Laufwerksbezeichnung D vorhanden sein, die mehr freie Speicherkapazität besitzt. Zudem ist möglicherweise als Laufwerk E eine RAM-Disk eingerichtet. Obwohl die Zwischenspeicherung auf einer RAM-Disk schneller als auf einer Diskette oder einer Festplatte ist, kann deren Verwendung vom Benutzer nicht gewünscht sein, da die Inhalte beim Abschalten des Systems verlorengehen. Es gibt also gute Gründe für die Verwendung logischer Laufwerke und Verzeichnisse anstelle konkreter physikalischer Angaben wie A:\ oder C:\TEMP. Der Benutzer sollte unabhängig vom Anwendungsprogramm festlegen können, welchem logischen Laufwerk oder Verzeichnis welche physikalische Einheit jeweils zugeordnet ist.

Ein weiteres Beispiel ist ein Rechtschreibprüfungsprogramm, das zwei Wörterbücher auf einem Datenträger ablegt. Dabei wird angenommen, daß die Wörterbuchdateien bei der Installa-

tion des Programms auf eine Festplatte kopiert wurden. Welches Laufwerk und Verzeichnis aber enthält die Dateien tatsächlich? Der Entwickler des Rechtschreibprogrammes könnte ein Verzeichnis wie C:\RESCHR\DIKT1 im Programm festlegen. Falls ein Benutzer aber kein Laufwerk C besitzt oder die Kapazität dieses Laufwerks nicht ausreicht, kann das Rechtschreibprogramm mit einer solchen Festlegung nicht auf Laufwerk D installiert werden. Wie kann ein Programm dem Benutzer die Flexibilität zur Speicherung von Wörterbuchdateien an einer beliebigen Stelle bieten?

Die Antwort auf dieses Problem liefert die Verwendung logischer Einheiten- und Verzeichnisnamen. Hierbei kommt es auf drei Merkmale an:

- Der Benutzer darf logische Einheiten und Verzeichnisse beliebigen physikalischen Einheiten und Verzeichnissen zuordnen. Die Zuordnungen sollten veränderbar sein, ohne daß darauf zugreifende Programme davon Kenntnis haben müssen.
- Die zur Verfügung stehenden logischen Einheiten und Verzeichnisse sollten groß und quasi beliebig erweiterbar sein. Es gibt immer wieder neue Programme, wie etwa das für eine Rechtschreibüberprüfung, die ein neues logisches Verzeichnis benötigen.
- Bei der Namensgebung sollte eine möglichst große Freiheit gegeben sein, ohne Kollisionen zu begünstigen. Bei einem eng begrenzten Zeichensatz, aus dem die Namen zusammengestellt werden müssen, (wie *X1*, *X2* und *X3*) kommt es unweigerlich zu Mehrfachnennungen in unterschiedlichen Programmen.

Die Originalversion von MS-DOS stellte keine logischen Einheiten- und Verzeichnisnamen bereit. Zur Zeit der Entwicklung dieses Betriebssystems bestand die maximale PC-Konfiguration aus zwei Diskettenlaufwerken mit 64 Kbyte Speicherplatz. Der Benutzer mußte *Disk Jockey* spielen, und nur er konnte festlegen, in welchem Laufwerk die Programm- und Datendisketten

zu liegen hatten. Logische Einheitenbezeichnungen waren unter diesen Umständen nicht von Nutzen. Logische Verzeichnisse wurden unter MS-DOS 1.0 ebenfalls nicht benötigt, da in dieser Version die Verwendung von Unterverzeichnissen nicht unterstützt wurde. In den späteren DOS-Versionen 2.X und 3.X wurde an den physikalischen Bezeichnungen festgehalten, zum einen aufgrund der engen Speichergrenzen, zum anderen aus Gründen der Kompatibilität zu MS-DOS 1.0.

Keiner dieser Gründe gilt für OS/2. Alle Anwendungsprogramme, die im *Protect*-Modus von OS/2 ablaufen sollen, müssen umgeschrieben werden. OS/2 hat Zugriff auf umfangreiche Speicherkapazität. Zugleich bietet sich die Verwendung logischer Einheiten- und Verzeichnisnamen an: Alle OS/2-Systeme besitzen Festplatten oder ähnliche Speichereinheiten und es werden auf den OS/2-Systemen im allgemeinen eine Vielzahl hochentwickelter Anwendungen ausgeführt, die Zugriff auf lokale Dateien und Arbeitsbereiche benötigen. Aus diesen Gründen wurden die von MS-DOS bekannten Umgebungsstrings in OS/2 erweitert, so daß logische Namensbezeichnungen verwendet werden können.

Unter MS-DOS ist die Umgebungsliste bedingt durch die Art der Speicherbelegung und die fehlende Segmentverschiebung und -auslagerung im *Real*-Modus stark begrenzt. OS/2 ermöglicht ein fast beliebiges Wachstum der Umgebungssegmente, lediglich die 64-Kbyte-Begrenzung durch die Hardware ist zu beachten. Unter MS-DOS gibt es nur ein Umgebungssegment für das gesamte System. Falls ein Kindprozeß den Umgebungsstring verändert, gilt diese Änderung auch für den Vaterprozeß. Unter der OS/2-Architektur besitzt jeder Prozeß ein eigenes Umgebungssegment. Standardmäßig erbt der Kindprozeß eine Kopie des Segmentes des Vaterprozesses, der Vaterprozeß kann aber die Werte der Umgebung bei der Ausführung von *DosExecPgm* verändern.

Die Verwendung der Umgebungsstrings zur Bereitstellung logischer Namensbezeichnungen bietet sich an. Falls für die verwendeten logischen Bezeichnungen noch keine Konvention be-

steht, sollten bedeutungsvolle Namen gewählt werden. Die Installationsanweisung oder -software sollte den Einsatz der Umgebungsstrings dokumentieren. Eine Anwendung sollte eine Fehlermeldung oder einen Standardvorschlag anzeigen, wenn logische Namen im Umgebungsstring nicht erscheinen. Da jeder Prozeß ein eigenes Umgebungssegment von seinem Vaterprozeß erbt, können Stapelverarbeitungsdateien und Initialisierungsprogramme, die Anwendungen laden, die benötigten Umgebungsstrings bereits setzen. Auf diese Weise können mehrere Anwendungen und mehrere Kopien der gleichen Anwendung denselben logischen Namen unterschiedlich definieren.

Folgende Umgebungsvariablen sind üblich:

PATH=

PATH definiert eine Verzeichnisliste, in der CMD.EXE nach der auszuführenden Programmdatei sucht. Die einzelnen Verzeichnisnamen sind, durch Semikola getrennt, in die Liste aufzunehmen. Die Verzeichnisse werden in der Reihenfolge der Liste von links nach rechts durchsucht. Der folgende Befehl bedeutet, daß C:\BIN vor D:\TOOLS und erst anschließend das aktuelle Verzeichnis durchsucht wird:

```
PATH=C:\BIN;D:\TOOLS;
```

DPATH=

DPATH definiert eine Liste der Verzeichnisse, in denen ein *DosOpen*-Aufruf nach Datendateien sucht. Die Verzeichnisse werden durch Semikola getrennt. Die Liste wird von links nach rechts abgearbeitet, und die Verzeichnisse werden in dieser Folge durchsucht:

```
DPATH=C:\DBM;D:\TEMP;
```

Die Verwendung von DPATH erleichtert die Arbeit: Man kann aus einem Verzeichnis heraus arbeiten und Datendateien in einem anderen Verzeichnis ansprechen (das in DPATH aufgeführt wird), ohne daß der vollständige Pfadname angegeben werden muß. Offensichtlich ist allerdings, daß Anwendungen und Benutzer diese Methode mit Bedacht einsetzen müssen. Die um-

fangreiche Suche nach einem Dateinamen ist nicht ungefährlich, da die falsche Datei gefunden werden könnte (gleiche Dateinamen können in verschiedenen Verzeichnissen eines Systems häufig vorkommen). Zur Verwendung des DPATH-Strings muß ein Anwendungsprogramm den *DosScanEnv*-Aufruf zur Lokalisierung des DPATH-Strings und anschließend *DosSearchPath* zur Suche nach der Datendatei aufrufen.

INCLUDE=

Der INCLUDE-String definiert das Laufwerk und Verzeichnis, in dem die INCLUDE-Dateien für Compiler und Assembler standardmäßig abgelegt sind.

INIT=

Die INIT-Zeichenkette definiert das Laufwerk und Verzeichnis, das Initialisierungs- und Konfigurationsinformationen enthält. So kann ein Anwendungsprogramm beispielsweise eine Datei erstellen, die die vom Benutzer bevorzugten Standardwerte enthält. Derartige Dateien können in dem durch INIT angegebenen Verzeichnis gespeichert werden.

LIB=

Der LIB-String definiert das Laufwerk und Verzeichnis, in dem die Standardbibliotheksmodule gespeichert werden.

PROMPT=

Die Zeichenkette PROMPT legt das Erscheinungsbild der von CMD.EXE angezeigten Eingabeaufforderung fest. Besondere Zeichensequenzen sind vordefiniert, so daß CMD.EXE als Eingabeaufforderung das Arbeitsverzeichnis sowie Datum und Zeit anzeigen kann. In der Dokumentation zu CMD.EXE finden Sie weitere Details.

TEMP=

Der TEMP-String definiert das Laufwerk und Verzeichnis für Temporärdateien. Dieses Verzeichnis sollte sich auf einer relativ *schnellen* Speichereinheit befinden und genügend Kapazität für die Zwischenspeicherung von Dateien aufweisen. Das TEMP-

Verzeichnis sollte als *flüchtig* angesehen werden, dessen Inhalt bei jedem Systemstart verloren geht.

Mit dem Umgebungssegment läßt sich ein System flexibel an ein oder mehrere Anwendungsprogramme anpassen. Beispielsweise können auf diese Weise Standardvorgaben für Applikationen festgelegt werden.

Kapitel 11

Kommunikation zwischen Prozessen

Die Kommunikation zwischen Prozessen (IPC für *interprocess communication*) stellt einen zentralen Punkt im Konzept von OS/2 dar. Wie bereits erläutert, wird eine effektive Kommunikation zwischen den Prozessen zur Unterstützung der tool-basierten Architektur benötigt. Gleiches gilt für die Dynlink-Schnittstellen für Dienstleistungsroutinen (die zwischen Prozessen bereitgestellt werden). OS/2 stellt verschiedene Formen zur Erfüllung der Kommunikationsanforderungen zur Verfügung.

11.1 Gemeinsam benutzter Speicher

Der gemeinsam benutzte Speicher (*shared memory*) wurde bereits detailliert erläutert. Eine kurze Zusammenfassung: Es gibt zwei Arten von gemeinsam genutzten Speicherbereichen, den *benannten gemeinsam benutzten Speicher* (*named shared memory*; der Zugriff wird vom Klientenprozeß über den Namen des Speicherbereichs angefordert) und den *Übergabespeicher* (*give-away shared memory*; der Besitzer des Speicherbereichs kann den Zugriff an einen anderen Prozeß übergeben). Der gemeinsam verwendete Speicherbereich ist die effizienteste Form der IPC, da keine Kopiervorgänge oder Aufrufe des Betriebssystemkerns benötigt werden, nachdem der Speicherbereich einmal initialisiert ist. Gemeinsam benutzter Speicher verlangt allerdings einen größeren Aufwand bei der Benutzung durch Prozesse. Es muß ein Protokoll erstellt werden, üblicherweise werden auch Semaphoren und Flaggen benötigt. Außerdem müssen die Möglichkeiten fehlerhafter Programme und der vorzeitigen Beendigung von Prozessen in die Überlegungen einfließen. An-

wendungsprogramme mit einem geringen Datenvolumen sollten auf *benannte Pipes* zurückgreifen.

11.2 Semaphoren

Eine Semaphore ist eine Flagge oder ein Signal (nicht im Sinne der Signalfähigkeit von OS/2). In der Grundform kann eine Semaphore nur zwei Modi annehmen: Aktiv oder Desaktiv (ein oder aus). Eine Semaphore ist ein Signal, das einen Thread zur Signalisierung eines Zustandes verwendet (Beispiel: Signal für die Belegung von Systemressourcen).

OS/2 unterstützt zwei Semaphorentypen, von denen wiederum jede auf zweierlei Weisen genutzt werden kann. Die zwei Typen von Semaphoren - *RAM-Semaphoren* und *Systemsemaphoren* - haben viel gemeinsam und es wird dieselbe System-API (*application program interface*) zur Manipulation verwendet. Eine RAM-Semaphore ist (wie ihr Name impliziert) eine 4-byte-Datenstruktur im Hauptspeicher (RAM). Die Datenstruktur muß allen Threads zugänglich sein, die mit ihr arbeiten. Die System-API, die RAM-Semaphoren manipuliert, ist im Dynlink-Subsystem zu finden. Der Code fordert Semaphoren mit einer Testen-und-Setzen-Operation an, die nicht unterbrochen werden kann. Es muß nicht der Kernel (Ring 0) angesprochen werden, um gegen einen vorzeitigen Abbruch zu sichern. Die einfachsten Tasks - Anforderung oder Freigabe einer Semaphore, auf die keine Threads warten - sind folglich sehr schnell auszuführen. Die benötigte Zeit beläuft sich auf 100 Millisekunden auf einem IBM PC/AT mit sechs MHz Taktfrequenz und 1-wait-state-Speicher (für Geschwindigkeitsmessungen ist dies die Standardumgebung, die den denkbar schlechtesten Fall darstellt, da es wohl keinen Personal Computer gibt, der langsamer arbeitet). Falls eine Semaphore bereits angefordert ist und das aufrufende Programm eine Blockierung vornimmt oder falls ein weiterer Thread auf die Semaphore wartet, muß das Semaphoren-Dynlink-Paket in den *Kernel*-Modus umschalten.

Auf der anderen Seite verwenden Systemsemaphoren Datenstrukturen, die im Systemspeicher gelagert sind und außerhalb

des Adressierungsbereichs der Prozesse liegen. Aus diesem Grund sind Systemsemaphorenoperationen langsamer als RAM-Semaphorenoperationen. Für eine unangefochtene Semaphorenanforderung muß man mit 350 Mikrosekunden rechnen. Einige Vorteile machen diesen Geschwindigkeitsverlust aber wieder wett. Systemsemaphoren unterstützen Mechanismen, die einen *Deadlock* durch Programmabbruch verhindern; auch Exklusivitäts- und Counting-Leistungsmerkmale werden unterstützt. Als allgemeine Regel kann gelten, daß RAM-Semaphoren eingesetzt werden sollten, wenn eine Anforderung nur einen Prozeß betrifft. Bei mehreren Prozessen sollten Systemsemaphoren verwendet werden.

Der erste Schritt im Umgang mit Semaphoren ist in jedem Fall das Anlegen einer Semaphore. Ein Anwendungsprogramm erzeugt eine RAM-Semaphore durch die Belegung eines 4-byte-Bereichs, der auf den Wert 0 initialisiert wird. Die *far*-Adresse des Bereiches ist die Nummer der RAM-Semaphore. Ein *DosCreateSem*-Aufruf erzeugt Systemsemaphoren. Beim Aufruf wird ein Argument zur Festlegung der Exklusivität übergeben. Obwohl die Semaphoren die Ausführung von Threads steuern, gehören Semaphorennummern zu einem Prozeß. Nach dem Anlegen einer Semaphore und der Übergabe ihrer Nummer können alle Threads in diesem Prozeß diese Nummer verwenden. Andere Prozesse müssen die Semaphore über *DosOpenSem* öffnen. Für eine RAM-Semaphore gibt es kein explizites Öffnen. Zur Verwendung für die Kommunikation zwischen Prozessen (IPC) muß die RAM-Semaphore in einem gemeinsam verwendeten Speichersegment enthalten sein, so daß auch andere Prozesse darauf zugreifen können. Ein anderer Prozeß muß lediglich die *far*-Adresse der RAM-Semaphore kennen. Eine RAM-Semaphore wird initialisiert, indem deren 4-byte-Speicherbereich mit Nullen gefüllt wird.

Vom Öffnen und Schließen abgesehen verwenden RAM- und Systemsemaphoren dieselben OS/2-Semaphorenaufrufe. Jeder Semaphorenaufruf übernimmt eine Semaphorennummer als Argument. Die Nummer einer RAM-Semaphore ist gleichzeitig ihre Adresse. Die Nummer einer Systemsemaphore wurde beim

Anlegen- oder Öffnungsaufruf zurückgegeben. Die OS/2-Semaphorenroutinen unterscheiden zwischen RAM- und Systemsemaphoren durch die Untersuchung der übergebenen Nummer. Da Systemsemaphoren und deren Namen in einem internen Datenbereich von OS/2 gehalten werden, stehen davon nur eine bestimmte Anzahl zur Verfügung. Die Anzahl der RAM-Semaphoren ist lediglich durch den verfügbaren Hauptspeicher begrenzt.

Die häufigste Verwendung von Semaphoren ist der Schutz kritischer Bereiche. Zur Wiederholung: Ein kritischer Bereich ist ein Programmteil, der Datenressourcen in einer eintrittsinvarianten Weise (nicht simultan durch verschiedene Programmteile benutzbar) manipuliert. Man kann auch sagen, daß die Ausführung eines kritischen Programmbereichs beim gleichzeitigen Aufruf von zwei Threads, die dieselben Datenressourcen bearbeiten, zu Problemen führt. Ein kritischer Bereich kann mehr als einen Programmteil umfassen. Falls eine Unterroutine Einträge in eine Tabelle einfügt und eine weitere Unterroutine Einträge löscht, sind beide Routinen ein kritischer Bereich in Bezug auf die Tabelle. Ein Vergleich für einen kritischen Bereich ist beispielsweise ein Waschraum in einem Flugzeug; die Semaphore ist das Zeichen, das von außen sichtbar ist und *belegt* anzeigt. Der erste Benutzer setzt die Semaphore und beginnt mit der Datenmanipulation. Wenn zwischenzeitlich andere Benutzer ankommen, sehen diese, daß die Semaphore besetzt ist und blockieren (das heißt, warten auf den Zugriff) die Semaphore. Wenn der kritische Bereich wieder verfügbar wird, wird die Semaphore gelöscht, und lediglich einer der wartenden Threads kann darauf zugreifen. Die anderen warten weiterhin.

Die Verwendung von Semaphoren zum Schutz von kritischen Bereichen ist unkompliziert. Am Beginn eines Programmteils, der der Bearbeitung kritischer Daten dient, wird ein *DosSemRequest*-Aufruf eingefügt. Nach der Bearbeitung des Aufrufs ist die Semaphore für den aktuellen Thread belegt und das Programm kann weiter abgearbeitet werden. Nach der Beendigung des kritischen Programmteils wird ein *DosSemClear*-Aufruf

ausgeführt. *DosSemClear* gibt die Semaphore frei und reaktiviert einen der wartenden Threads.

Systemsemaphoren unterscheiden sich von RAM-Semaphoren in einem wichtigen Aspekt. Wird eine Systemsemaphore für die exklusive Benutzung eines Bereichs angelegt, kann sie als Zählsemaphore eingesetzt werden. *Exklusive Benutzung* meint hier, daß nur der Thread, der die Semaphore gesetzt hat, sie wieder löschen kann. Dies ist eine Abweichung vom Prinzip des Ressourcenbesitzes von Prozessen. Der Thread (nicht der Prozeß) besitzt das Privileg zum Löschen der für die exklusive Benutzung angelegten Semaphore. Eine Zählsemaphore kann beliebig oft gesetzt werden, es muß aber die gleiche Anzahl von Freigabeaufrufen erfolgen, bevor die Zählsemaphore tatsächlich freigegeben wird. Stellen Sie sich eine Anwendung vor, die eine Funktion A und eine Funktion B besitzt, von denen jede einen kritischen Bereich verändert. Jede Funktion fordert eine Semaphore zu Beginn der Ausführung an und gibt sie nach Beendigung der Aufgabe wieder frei. Unter bestimmten Umständen muß die Funktion A die Funktion B aufrufen. A kann die Semaphore vor dem Aufruf von B nicht freigeben, da sie sich im kritischen Bereich befindet und die Daten in einem inkonsistenten Zustand vorliegen. Wenn B einen *DosSemRequest*-Aufruf für die Semaphore ausgäbe, blockiert dies die Bearbeitung, da die Semaphore bereits von Funktion A benutzt wird.

Die Zählsemaphore löst dieses Problem. Wenn die Funktion B einen zweiten (redundanten) *DosSemRequest*-Aufruf ausführt, erkennt OS/2 dies als denselben Thread, der bereits die Semaphore angefordert hat und erhöht den Zähler der Semaphore um 1. Gibt Funktion B die Semaphore wieder frei, dekrementiert OS/2 den Zähler um den Wert 1. Da der Zähler noch nicht auf dem Wert 0 steht, wird die Semaphore nicht tatsächlich freigegeben. Sie wird erst dann wieder systemweit verfügbar, nachdem Funktion B die Kontrolle an Funktion A zurückgegeben und letztere beendet wurde (dabei wird die Semaphore ein zweites Mal freigegeben).

Eine zweite wichtige Verwendung für Semaphoren liegt in ihrer Funktion als Signalgeber (nicht zu verwechseln mit der Signaleinrichtung von OS/2). Semaphoren werden eingesetzt, um Threads das Auftreten bestimmter Ereignisse zu signalisieren. Betrachten wir eine Anwendung, in der mehrere Threads abgearbeitet werden. Ein Thread wird zur Kommunikation über den seriellen Port und ein weiterer Thread zur Berechnung von Ergebnissen, die sich aus der Kommunikation ergeben, verwendet. Der Thread zur Berechnung veranlaßt den Kommunikations-Thread zum Aussenden einer Meldung und zum Empfang einer Antwort und führt anschließend weitere Aufgaben (zur Berechnung) aus. Zu einem späteren Zeitpunkt muß der Berechnungs-Thread den Kommunikations-Thread blockieren, sofern noch keine Antwort erhalten wurde - bei vorliegender Antwort soll die Blockierung entfallen.

Das Problem kann durch die Verwendung einer Semaphore als Flagge gelöst werden. Der Berechnungs-Thread setzt über einen *DosSemSet*-Aufruf die Semaphore "vor der Auftragserteilung" an den Kommunikations-Thread. Ist der Thread zur Übernahme der Antwort bereit, erfolgt ein *DosSemWait*-Aufruf mit der bereits gesetzten Semaphore. Beim Empfang der Antwort durch den Kommunikations-Thread wird die Semaphore gelöscht. Bei einem *DosSemWait*-Aufruf durch den Berechnungs-Thread können weitere Berechnungen ohne Verzögerung ausgeführt werden (sofern die Semaphore bereits gelöscht ist). In diesem Fall wird kein kritischer Bereich mehr geschützt. Die Semaphoren werden gesetzt oder gelöscht, um verschiedenen Threads ein Ereignis mitzuteilen. Die Notwendigkeiten sind hier denen einer "Kritischen-Bereich-Semaphore" entgegengesetzt: die Semaphore soll nicht exklusiv benutzt werden, sondern anderen Threads zugänglich sein. Wäre dies der Fall, könnte der Kommunikations-Thread die Semaphore nicht freigeben. Die Semaphore soll auch nicht als Zähler verwendet werden. Falls eine Zählung vorgenommen würde, erfolgte durch einen *DosSemWait*-Aufruf des Berechnungs-Threads keine Blockierung und OS/2 würde erkennen, daß der *DosSemSet*-Aufruf bereits zuvor ausgeführt wurde und den Semaphorenzähler erhöhen.

OS/2 verwendet Semaphoren zur Signalübermittlung bei der asynchronen Kommunikation, um die Beendigung einer Ein-/Ausgabeoperation anzuzeigen. Die Systemzeitgeberroutine setzt Semaphoren ein, um das Verstreichen einer gegebenen Zeitspanne anzuzeigen. OS/2 unterstützt eine Sonderform des Wartens auf Semaphoren (*DosMuxSemWait*), wodurch ein Thread zu einem Zeitpunkt auf mehr als eine Semaphore warten kann. Sobald alle spezifizierten Semaphoren gelöscht wurden, kehrt *DosMuxSemWait* zurück. Der *DosMuxSemWait*-Aufruf (wie *DosSemWait*) wartet darauf, daß Semaphoren gelöscht werden. Die Semaphore wird nicht wie mit einem *DosSemRequest*-Aufruf gesetzt oder angefordert.

11.2.1 Wiedergewinnung von Semaphoren

Die Schwierigkeiten, die entstehen können, wenn der aufrufende Prozeß einer gesetzten Semaphore in unerwarteter Weise beendet wird und die Semaphore somit "verwaist", wurden bereits erwähnt. Dennoch soll dieser Punkt hier nochmals angesprochen werden, da die Behandlung dieser Situation durch Anwendungsprogramme kritisch ist und es sehr schwierig ist, zeitbezogene Fehler im Rahmen von Programmtests aufzuspüren.

Semaphoren können auf mindestens vier Arten verwaisen:

- Ein ankommendes Signal veranlaßt die CPU zur Unterbrechung des Arbeitsablaufes, und bei der signalbearbeitenden Routine tritt ein Fehler beim Rücksprung zum Unterbrechungspunkt auf.
- Ein Prozeß beendet einen anderen Prozeß ohne Warnung.
- Ein Prozeß führt zu einem schweren GP-Fehler.
- Ein Prozeß wird aufgrund eines Programmierfehlers in einer unerwünschten Weise ausgeführt und die Semaphore wird dabei nicht mehr freigegeben.

Die in diesen Fällen auszuführenden Reaktionen sind von der Verwendung der Semaphoren abhängig. In einigen Situationen (Beispiel: Berechnungs- und Kommunikations-Thread) wird

keine explizite Reaktion benötigt. Bei der Beendigung des Prozesses wird auch die Semaphore gelöscht. Eine spezielle Behandlung ist lediglich nötig, wenn die Anwendung einen *DosExitList*-Aufruf zur Ausführung des Programms, das die Semaphore verwendet, benutzt. Dies sollte nur selten geschehen, da Semaphoren in einem Prozeß zur Koordination verschiedener Threads eingesetzt werden und bei der Aktivierung der *Exit*-Liste nur ein Thread übrig bleibt. In ähnlicher Weise kann ein Prozeß Signale nur empfangen, wenn diese angefordert wurden. Eine Anwendung, die nicht mit Signalen arbeitet, braucht die Unterbrechung eines kritischen Bereichs durch Signale nicht zu befürchten. Applikationen mit Signalen können den *DosHoldSignal*-Aufruf verwenden, von der signalbearbeitenden Routine zurückkehren oder Thread 1 (Signalbehandlungs-Thread) umgehen, indem sie nicht in kritische Bereiche vordringen.

In anderen Situationen kann eine Semaphore wiedergewinnbare Ressourcen schützen. So kann beispielsweise eine Systemsemaphore zum Schutz eines Druckers verwendet werden, der aus irgendeinem Grund direkt durch Anwendungen und nicht über den System-Spooler angesprochen wird. Wird der aktuelle Eigner der Systemsemaphore *Druckerverwendung* unerwartet beendet, erhält der nächste auf die Semaphore wartende Thread den Zugriff und einen Fehlercode, der aussagt, daß der Semaphorenbesitzer beendet wurde, ohne die Semaphore zuvor freizugeben. In diesem speziellen Fall kann die Anwendung einfach einen Seitenvorschub auslösen und mit dem Druck beginnen. Andere mögliche Reaktionen sind die Säuberung der geschützten Ressource oder die Ausführung eines Prozesses, der diese Aufgabe erledigt. Eine Anwendung kann dem Benutzer auch eine Meldung wie *Die Datenbank ist inkonsistent, entsprechende Maßnahmen durchführen!* ausgeben und die Kontrolle an die Betriebssystemebene übergeben. In diesem Fall sollte die Anwendung bewußt vorzeitig beendet werden, ohne zuvor die Semaphore freizugeben. Andere wartende Threads erhalten die entsprechende *Besitzer-beendet*-Meldung. Nachdem dieser Code einmal übergeben wurde, wird der Fehlerstatus gelöscht (falls der dem fehlerhaften Programm folgende Thread die Semaphore

ohne Korrektur im kritischen Bereich freigibt, wird dies zum Problem führen).

Falls ein Prozeß die eigenen Semaphoren über eine *DosExitList*-bearbeitende Routine beenden soll, sind weitere Überlegungen nötig. Zuerst einmal müssen exklusive (das heißt Zähl-) Semaphoren benutzt werden. Eine *ExitList*-Routine kann feststellen, daß eine RAM- oder nicht-exklusive Systemsemaphore reserviert ist. Nicht feststellbar ist, ob die Semaphore von diesem Prozeß reserviert wurde. Der Programmierer könnte auf den Gedanken kommen, ein Flaggenbyte zu verwenden, das bei der Anforderung der Semaphore gesetzt und bei der Freigabe wieder gelöscht wird. Das Löschen enthält aber eine potentielle Gefahrenquelle, wenn die Flagge vor einem *DosSemRequest*-Aufruf gesetzt wird. In diesem Fall könnte der Thread zwischen dem Setzen der Flagge und dem Empfangen der Semaphore beendet werden. Die *Exit*-Routine nimmt fälschlicherweise an, daß die Semaphore nun dem aktuellen Prozeß angehört und gibt sie frei - eine unerwünschte Überraschung für den wahren Besitzer der Semaphore. Umgekehrt kann ein Thread eine Semaphore anfordern und anschließend eine Flagge setzen. Aber auch hier besteht die Gefahr, daß der Thread zwischen der Anforderung der Semaphore und dem Setzen der Flagge beendet wird. Auch dieser Zustand führt zu schwerwiegenden Fehlern.

Die Verwendung von exklusiven Systemsemaphoren löst die Probleme. Wie bereits erwähnt, wird die Systemsemaphore eines vorzeitig beendeten Threads in einen speziellen *Besitzer-beendet*-Status versetzt, so daß dem nächsten darauf zugreifenden Thread der Verwaisungsstatus mitgeteilt werden kann. Bei exklusiv verwendeten Systemsemaphoren gibt es einen weiteren Punkt zu beachten: Sollte der Prozeß aufgrund eines externen Ereignisses oder eines *DosExit*-Aufrufs beendet werden und eine *DosExitList*-bearbeitende Routine enthalten, werden alle verwaisten Systemsemaphoren in einen besonderen *Besitzer-beendet*-Status versetzt, so daß nur der verbleibende Thread des Prozesses (der die *DosExitList*-bearbeitende Routine ausführt) die Semaphore anfordern kann. Bei der Ausführung erhält der Thread den *Besitzer beendet*-Sondercode. Die die *Exit*-Liste be-

arbeitende Routine kann einen *DosSemWait*-Aufruf mit einem Timeout-Wert von 0 verwenden, um festzustellen, ob die Semaphore gesetzt ist. Bei der Rückgabe des *Besitzer-beendet*-Codes löscht die *DosExitList*-Routine die Ressource und führt ein *DosSemClear*-Aufruf zum Löschen der Semaphore aus. Bei der Beendigung eines Threads durch einen expliziten *DosExit*-Aufruf mit dem *Beende-diesen-Thread*-Code werden alle Systemsemaphoren zur exklusiven Verwendung nicht den besonderen *Besitzer-beendet*-Status erlangen. Statt dessen wird der allgemeine *Besitzer-beendet*-Status zugeordnet, der jedem Thread des Systems die Anforderung der Semaphore und das Empfangen des Codes ermöglicht. In gleicher Weise werden alle Semaphoren mit dem besonderen *Besitzer-beendet*-Status, die von der *DosExitList*-bearbeitenden Routine nicht gelöscht wurde, zu Semaphoren mit dem normalen *Besitzer-beendet*-Status, wenn der Prozeß vollständig beendet wird.

11.2.2 Zuordnung der Semaphoren

Obwohl mehrere Threads auf eine Semaphore warten können, kann jede Semaphore zu einem Zeitpunkt nur einem Thread zur Verfügung stehen. OS/2 verteilt den Semaphorenzugriff auf derselben Basis wie die CPU-Priorität: Der wartende Thread mit der höchsten Priorität erhält eine freiwerdende Semaphore. Falls mehrere Threads mit der höchsten Priorität warten, verteilt OS/2 die Zugriffsberechtigung für die Semaphore gleichmäßig.

11.3 Benannte Pipes

Das System der anonymen Pipes - stromorientierte IPC-Mechanismen (das sind Mechanismen, die mit Datenströmen arbeiten), die über *DosRead*- und *DosWrite*-Aufrufe funktionieren - wurde bereits erläutert. Zwei Prozesse können über anonyme Pipes nur dann kommunizieren, wenn der eine Prozeß ein Nachfolgeprozeß des anderen ist und der Nachfolgeprozeß die Nummer der Pipe vom Vaterprozeß geerbt hat. Anonyme Pipes werden fast ausschließlich zur Übertragung von Ein- und Ausgabedaten

von und zu Kindprozessen und von und zu Unterbäumen der Kindprozesse verwendet.

OS/2 unterstützt eine weitere Form der Pipes - die *benannten Pipes*. Benannte Pipes stehen in OS/2 Version 1.0 noch nicht zur Verfügung, wohl aber in künftigen Versionen. Benannte Pipes ähneln den anonymen Pipes insofern, als sie einen seriellen Kommunikationskanal zwischen zwei Prozessen darstellen und die *DosRead-/DosWrite*-Schnittstelle verwenden. Andererseits gibt es einige Punkte, in denen sie Unterschiede aufweisen:

- Benannte Pipes besitzen Namen im Dateisystemnamensbereich. Prozesse, die eine benannte Pipe benutzen, müssen nicht miteinander in Beziehung stehen, lediglich der Name der Pipe muß für den Zugriff bekannt sein.
- Da benannte Pipes den Dateisystemnamensbereich verwenden und der Namensbereich Geräte in einem Netzwerk beschreiben kann, können benannte Pipes sowohl lokal (innerhalb eines Gerätes) als auch im *Remote*-Zugriff (in einem Netzwerk) verwendet werden.
- Eine anonyme Pipe ist ein byte-orientierter Mechanismus. Das System betrachtet die durch eine anonyme Pipe gesendeten Daten als nichtdifferenzierten Strom von Bytes. Der schreibende Prozeß kann einen 100-byte-Datenblock aussenden und der lesende Prozeß kann zwei 30-byte-Blöcke und einen 40-byte-Block empfangen. Falls der Bytestrom einzelne Meldungen enthält, muß das empfangende Element bestimmen können, wo die Meldungen beginnen und enden. Benannte Pipes können im Bytestrommodus verwendet werden, sie unterstützen aber auch einen Meldungsmodus, in dem Prozesse Ströme von Meldungen lesen und schreiben. Falls eine benannte Pipe im Meldungsmodus betrieben wird, separiert OS/2 (so kann man sich dies vorstellen) automatisch eine ankommende Meldung, so daß der lesende Prozeß nach der nächsten Meldung und nicht nach den nächsten 100 Bytes fragen muß.

- Benannte Pipes sind vollduplex, während anonyme Pipes ein Paar darstellen, bei dem jede Pipe halbduplex arbeitet. Bei der Anlage einer anonymen Pipe werden zwei Nummern zurückgegeben - eine Lese- und eine Schreibnummer. Das Öffnen einer benannten Pipe liefert eine einzige Nummer, die sich (abhängig vom *DosOpen*-Modus) sowohl auf das Lesen als auch auf das Schreiben beziehen kann. Obwohl eine benannte Pipe (vollduplex) über eine einzige Nummer angesprochen wird, sind die in verschiedene Richtungen transferierten Daten vollständig voneinander getrennt. Eine benannte Pipe sollte als zwei separate Pipes zwischen dem lesenden und dem schreibenden Prozeß betrachtet werden - eine Pipe, die ankommende Daten entgegennimmt und eine andere, die ausgehende Daten übergibt. Falls beispielsweise ein Thread über die Nummer einer benannten Pipe Daten in die Pipe schreibt und anschließend aus der Pipe über dieselbe Nummer Daten liest, sind dies nicht die Daten, die vom Prozeß selbst dorthin geschrieben wurden. Die vom Thread geschriebenen Daten befinden sich auf der Ausgangsseite, während die vom Thread gelesenen Daten auf der Eingangsseite vorliegen.

- Benannte Pipes werden häufig zur Kommunikation zwischen Prozessen eingesetzt, die Serviceleistungen an einen oder mehrere Klientenprozesse (üblicherweise gleichzeitig) liefern. Die benannte Pipe-API enthält Unterstützungsfunktionen: die Pipe-Wiederverwendbarkeit, mehrere Pipes mit identischen Namen usw. Die einzelnen Punkte werden noch erläutert.

- Benannte Pipes unterstützen Transaktionsein-/-ausgabeaufrufe, die eine effiziente Möglichkeit zur Implementierung lokaler und netzwerkübergreifender Prozeduraufrufdialoge zwischen Prozessen darstellen.

- Programme, die auf einer Workstation unter MS-DOS Version 3.X ablaufen, können auf benannte Pipes auf einem OS/2-Server zugreifen, um einen Dialog mit Server-An-

wendungen zu ermöglichen. Dies ist möglich, da sich ein Klient (eine benannte Pipe) wie eine Datei verhält.

Sie erinnern sich vielleicht, daß der eine anonyme Pipe anlegende Prozeß eine besondere Schnittstelle (*DosMakePipe*) verwenden muß. Ein Klientenprozeß kann aber die *DosRead*- und *DosWrite*-Funktionen mit der zurückgegebenen Nummer verwenden, ohne auf die Art des angesprochenen Elements Rücksicht zu nehmen. Dasselbe gilt für benannte Pipes, wenn diese im Strommodus verwendet werden. Der eine benannte Pipe anlegende Prozeß verwendet eine besondere API zur Initialisierung, dessen Klienten können die Pipe aber nutzen, ohne auf deren Natur einzugehen (solange es sich um eine serielle Verwendung handelt). Der wahlfreie Zugriff unter Verwendung des *DosSeek*-Aufrufes wird für Pipes nicht unterstützt und führt zur Rückgabe eines Fehlercodes. Benannte Pipes werden durch einen *DosMakeNmPipe*-Aufruf angelegt. Nach dem Anlegen der Pipe muß einer der Threads des Prozesses per *DosConnectNmPipe*-Aufruf auf das Öffnen der Pipe durch den Klienten warten. Ein Klient kann eine Pipe nicht erfolgreich öffnen, solange kein *DosConnectNmPipe*-Aufruf vom bedienenden Prozeß ausgeführt wurde.

Obwohl der bedienende Prozeß erkennt, daß eine benannte Pipe in Verwendung ist und aus diesem Grund eine besondere API für benannte Pipes aufruft, ist dies für den Klientenprozeß nicht der Fall, da ein normaler *DosOpen*-Aufruf zur Öffnung der Pipe eingesetzt wird. Da benannte Pipes im Dateisystemnamensbereich stehen, kann ein Klientenprozeß beispielsweise die "Datei" \PIPE\STATUS öffnen, ohne daß er weiß, daß es sich um eine benannte Pipe handelt, die von einem anderen Prozeß verwaltet wird. Der *DosMakeNmPipe*-Aufruf gibt eine Nummer an das "Server-Ende" der Pipe zurück, der *DosOpen*-Aufruf liefert eine Nummer an das "Klientenende". Sobald der Klientenprozeß eine Pipe öffnet, kehrt der *DosConnectNmPipe*-Aufruf zum Serverprozeß zurück.

Die Kommunikation über eine benannte Pipe ist der über eine anonyme Pipe ähnlich: Klient und Server geben Schreib- und

Leseaufrufe über die Nummer (entsprechend dem Öffnungsmodus) aus. Wenn ein Prozeß am einen Ende die Pipe schließt, erhält der Prozeß am anderen Ende einen Fehlercode als Antwort auf Schreiboperationen und eine EOF-Anzeige als Antwort auf Leseoperationen.

Das vorgestellte System ist recht einfach. In der Realität wird ein Serverprozeß üblicherweise auf den nächsten Klientenprozeß warten. Dies ist der Grund für den notwendigen *DosConnectNmPipe*-Aufruf. Nachdem der erste Klientenprozeß sein Ende der benannten Pipe geschlossen hat und das Server-Ende das EOF-Zeichen aus der Pipe erhält, gibt dieses einen *DosDisconnectNmPipe*-Aufruf zur Bestätigung aus, daß der Klient die Pipe geschlossen hat (explizit oder durch dessen Beendigung). Anschließend kann ein weiterer *DosConnectNmPipe*-Aufruf ausgeführt werden, um die Pipe zur erneuten Öffnung durch einen anderen oder denselben Klientenprozeß vorzubereiten. Der Verknüpfungs- und Freigabeprozeß erlaubt einem Server die Abarbeitung von mehreren Anforderungen mit einer einzigen benannten Pipe. Ein *DosDisconnectNmPipe*-Aufruf kann zur erzwungenen Beendigung der Kommunikation mit einem Klientenprozeß verwendet werden. Dies ist erforderlich, wenn ein Klientenprozeß eine ungültige Anforderung ausgibt oder andere Zeichen des Fehlverhaltens zeigt.

Die Bedienung mehrerer Klientenprozesse nacheinander ist auf die geschilderte Art und Weise möglich. Aber wie steht es mit der gleichzeitigen Bedienung von Anforderungen? Mit der beschriebenen Methode kann der Serverprozeß jeweils nur einen Klientenprozeß bedienen. Der *DosOpen*-Aufruf eines Klientenprozesses wird erfolglos beendet, wenn die benannte Pipe bereits von einem anderen Klientenprozeß benutzt wird oder der Serverprozeß noch keinen *DosConnectNmPipe*-Aufruf ausgeführt hat. An dieser Stelle kommt der *DosMakeNmPipe*-Aufruf ins Spiel, bei dem ein Parameter übergeben wird, den wir *Instanzparameter* nennen wollen.

Beim ersten Öffnen einer benannten Pipe wird der Instanzparameter im Flaggenwort der Pipe spezifiziert. Liegt er über 1,

kann die Pipe von einem Serverprozeß mehr als einmal geöffnet werden. Weitere Öffnungen werden mit *DosMakeNmPipe*-Aufrufen ausgeführt, die eine andere Nummer zum Zugriff auf eine neue Instanz der Pipe liefern. Offensichtlich wird die Pipe beim zweiten und nachfolgenden Aufrufen von *DosMakeNmPipe* nicht nochmals angelegt, aber der *DosOpen*-Aufruf kann nicht verwendet werden, da er das Klientenende der benannten Pipe (und nicht das Server-Ende) öffnet. Das Instanzzählargument wird für weitere Aufrufe der benannten Pipe ignoriert. Extrainstanzen einer benannten Pipe können von dem Prozeß, der auch die erste Instanz angelegt hat, oder von anderen Prozessen geschaffen werden. In Bild 11.1 sind mehrere Instanzen einer benannten Pipe gezeigt.

Hinweis: Wie andere residente benannte Objekte, die nicht dem Dateisystem unterliegen, bleibt eine benannte Pipe dem System bekannt, solange sie von mindestens einem Prozeß geöffnet ist. Wenn alle Nummern auf die benannte Pipe geschlossen werden, "vergißt" OS/2 alle Informationen über die benannte Pipe. Der nächste DosMakeNmPipe-Aufruf erzeugt eine neue benannte Pipe.

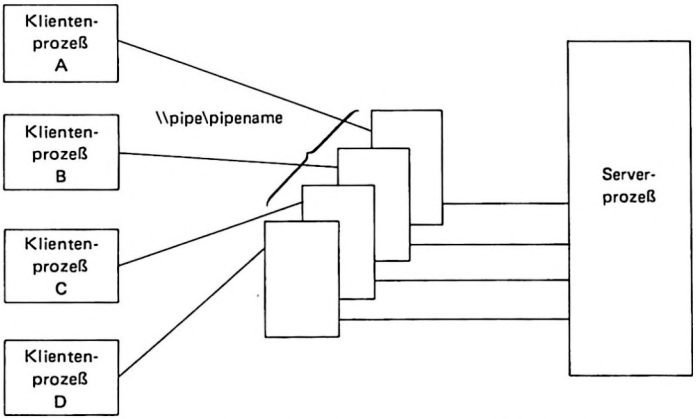


Bild 11.1 Mehrere Instanzen einer benannten Pipe

Bei der Ausführung eines *DosOpen*-Aufrufs durch einen Klientenprozeß auf eine benannte Pipe mit mehreren Instanzen verwendet OS/2 eine beliebige Server-Instanz der Pipe, für die ein *DosConnectNmPipe*-Aufruf ausgeführt wurde. Falls keine Instanz verfügbar oder empfangsbereit ist, wird dem Klientenpro-

zeß ein Fehlercode übergeben. OS/2 garantiert keine gerechte Verteilung ankommender Anforderungen für alle Server-Instanzen. Es wird angenommen, daß alle Server-Threads, die einen *DosConnectNmPipe*-Aufruf ausführten, gleichwertig sind.

Das Leistungsmerkmal mehrerer Instanzen erlaubt einem einzigen oder mehreren Serverprozessen die Bearbeitung mehrerer Klientenprozesse zur gleichen Zeit. Ein Prozeß, der vier Threads verwendet, kann vier Klientenprozesse so schnell bedienen wie vier Prozesse, die mit je einem Thread arbeiten. Solange die Threads sich nicht durch gegenseitiges Blockieren behindern, hat ein Mehrprozeß-Server keine Vor- oder Nachteile in der Bearbeitungseffizienz gegenüber einem Mehr-Thread-Server.

Das Paket zur Realisierung benannter Pipes von OS/2 enthält zwei Operationen für Klientenprozesse: *DosTransactNmPipe* und *DosCallNmPipe.*

DosTransactNmPipe ähnelt einem *DosWrite* gefolgt von einem *DosRead*: Es wird eine Meldung an das Server-Ende der benannten Pipe gesendet und anschließend die Antwort gelesen. Der *CallNmPipe*-Aufruf erledigt dasselbe mit einer nichtgeöffneten benannten Pipe und stellt quasi eine Kombination aus *DosOpen*-, *DosTransactNmPipe*- und *DosClose*-Aufrufen dar. Die Aufrufe sind von geringem Wert, solange Klienten- und Serverprozesse auf demselben Gerät ablaufen, der Klientenprozeß könnte durch die entsprechende Kombination von *DosOpen*-, *DosClose*-, *DosRead*- und *DosWrite*-Aufrufe dieselbe Aufgabe problemlos erledigen. In einer Netzwerkumgebung stellen diese Routinen aber eine wertvolle Arbeitshilfe dar. Falls der Serverprozeß auf einem anderen Rechner als der Klientenprozeß vorliegt, können OS/2 und die Netzwerkübertragung einen *datagram*-ähnlichen Mechanismus zur Implementierung dieser Aufrufe in einer für das Netzwerk effizienten Art einsetzen. Da benannte Pipes über ein Netzwerk transparent arbeiten, sollten alle Klientenprozesse, die diese Art der Operation ausführen, die vorgegebenen Aufrufe einsetzen (dies gilt auch, wenn der Programmentwickler nicht erwartet, daß sein Pro-

gramm in einer Netzwerkumgebung eingesetzt wird). Die Verwendung der Aufrufe stellt sicher, daß der Leistungsdurchsatz optimiert wird, wenn ein Benutzer sich zur Verwendung eines Serverprozesses innerhalb des Netzwerkes entschließt. Leser, die mit Netzwerkarchitekturen vertraut sind, erkennen den *DosCallNmPipe*-Aufruf als eine Form des Remote-Prozeduraufrufes. Es geht um den Prozeduraufruf eines Prozesses, der auf einem anderen Rechner vorliegt.

Die Einrichtungen für benannte Pipes umfassen eine Vielzahl von Funktionen, die die Bedeutung des tool-orientierten Ansatzes der OS/2-Architektur verdeutlichen. Das vorliegende Werk soll nicht in erster Linie die einzelnen Leistungsmerkmale von OS/2 im Detail beschreiben. Einige dieser Merkmale verdienen jedoch der besonderen Beachtung.

Die obigen Erläuterungen bezogen sich auf die stromorientierte Kommunikation, die einem Klientenprozeß die Verwendung einer benannten Pipe ermöglicht, ohne daß weitere Kenntnisse über die Pipe vorhanden sind. Sie können damit beispielsweise ein *Spooler*-Paket für eine vom System-Spooler nicht unterstützte Einheit entwickeln - beispielsweise für einen Plotter. Die Eingabe für den Spooler kann über eine benannte Pipe (vielleicht \PIPE\PLOT) erfolgen. Der Anwendung wird mitgeteilt, daß die Plotterausgabe an die Datei \PIPE\PLOT oder beispielsweise \\PLOTMACH\PIPE\PLOT (über ein Netzwerk) erfolgen soll und spricht auf diese Weise den Spooler am anderen Ende der benannten Pipe an.

Falls ein Klientenprozeß Kenntnis davon hat, daß eine benannte Pipe angesprochen wird und die ausgetauschten Informationen aus einer Serie von Meldungen und nicht aus einem Datenstrom bestehen, kann die benannte Pipe zur Übertragung von Meldungsströmen konfiguriert werden. In diesem Fall sind die Meldungen für die Schnittstelle nicht teilbar. Mit anderen Worten: Wenn ein Prozeß aus einer benannten Pipe liest, erhält er pro Lesevorgang eine Meldung (und zwar die vollständige Meldung). Meldungen können in einer Pipe in eine Warteschlange eingereiht werden, wobei OS/2 die Trennung zwischen den Meldun-

gen erkennt, so daß sie bei nachfolgenden Lesevorgängen korrekt aufgeteilt werden. Meldungsströme können in Netzwerkumgebungen effektiv eingesetzt werden, da die Netzwerkübertragung die Zusammenstellung der zu übertragenden Datenpakete in einer optimalen Weise vornimmt.

Benannte Pipes können auch in einer Weise konfiguriert werden, daß keine Blockierung stattfinden kann. Dies erlaubt einem Server- oder Klientenprozeß das Testen einer Pipe. Es kann festgestellt werden, ob die Pipe für eine bestimmte Operation bereit ist. So wird garantiert, daß der Prozeß nicht mit einer Wartezeit belastet wird. Prozesse können einen *DosPeekNmPipe*-Aufruf ausführen, ein ähnliches Leistungsmerkmal, das eine Probe der in der Pipe zum Lesen bereitstehenden Daten (ohne eigentliche Datenbearbeitung) verfügbar macht. Serverprozesse können den Aufruf verwenden, um Anforderungen von Klientenprozessen zu prüfen und zu bestimmen, ob sie zu diesem oder einem späteren Zeitpunkt bearbeitet werden sollen.

Es wurde schon erwähnt, daß bei dem Versuch, mit einem *DosOpen*-Aufruf zu öffnen, ein Fehlercode zurückgegeben wird, wenn zu diesem Zeitpunkt keine Instanz verfügbar ist. Üblicherweise wartet ein Klientenprozeß in dieser Situation darauf, daß der Service verfügbar wird. Dabei ist eine weitere Bearbeitung während der Wartezeit häufig unerwünscht. Für diese Situation wurde der *DosWaitNmPipe*-Aufruf bereitgestellt. Er erlaubt einem Klientenprozeß die Blockierung einer benannten Pipe, bis eine Instanz verfügbar wird. Nach der Rückkehr vom *DosWaitNmPipe*-Aufruf muß der Klientenprozeß einen *DosOpen*-Aufruf ausführen. Der *DosOpen*-Aufruf kann aber erfolglos sein, wenn ein anderer Prozeß die Pipe-Instanz in der Zeit zwischen dem Warte- und dem Öffnungsaufruf übernommen hat. Da mehrere wartende Elemente auf eine benannte Pipe in der Reihenfolge der Priorität bedient werden, ist dieser Systemzustand jedoch unwahrscheinlich.

11.4 Warteschlangen

Warteschlangen sind eine weitere Form des IPC. In gewisser Hinsicht gleichen sie benannten Pipes, sie unterscheiden sich aber auch von diesen. Wie benannte Pipes verwenden sie den Dateisystemnamensbereich, übergeben jedoch Meldungen statt Byteströme. Im Gegensatz zu benannten Pipes erlauben Warteschlangen das Aufnehmen mehrerer Schreibvorgänge in eine einzelne Schlange, da die Meldungen selbst Informationen über den sendenden Prozeß mit sich bringen. Der aus der Warteschlange lesende Prozeß kann zwischen Meldungen von unterschiedlichen Sendeprozessen unterscheiden. Benannte Pipes arbeiten strikt nach dem Prinzip des FIFO (*first in first out*), wohingegen in eine Warteschlange eingereihte Meldungen in beliebiger Reihenfolge gelesen werden können. Warteschlangen unterstützen gemeinsam verwendete Speicherbereiche als Übertragungsmechanismus, können aber nicht netzwerkweit arbeiten, obwohl sie bei der Übertragung umfangreicher Datenmengen schneller als benannte Pipes sind.

Die Schnittstelle des Warteschlangenpaketes ähnelt der Schnittstelle der benannten Pipes, ist mit dieser aber nicht identisch. Wie auch benannte Pipes, so besitzt jede Warteschlange einen Besitzer, von dem sie angelegt wurde. Klientenprozesse können die Warteschlange nun öffnen und schließen, während der Besitzer üblicherweise noch existiert. Im Gegensatz zu benannten Pipes müssen die Klientenprozesse eine *Queue*-API (*DosReadQueue*, *DosWriteQueue*) verwenden, die zum Einsatz mit dem Warteschlangenpaket entwickelt wurde. Obwohl jede Warteschlange nur einen Besitzer hat, können gleichzeitig mehrere Klienten vorhanden sein. Der Warteschlangenmechanismus benötigt kein Leistungsmerkmal, bei dem mehrere Warteschlangen desselben Namens angesprochen werden können und auch kein Äquivalent für den *DosWaitNmPipe*-Aufruf.

Meldungen in Warteschlangen unterscheiden sich von denen in benannten Pipes. Neben der eigentlichen Meldung enthält jede Warteschlangenmeldung zwei weitere Informationen. Die eine Information ist die PID des sendenden Prozesses, OS/2 stellt

diese bereit (der sendende Prozeß hat darauf keinen Einfluß). Die andere Information ist ein Wortwert, der vom Sender bereitgestellt wird und der von OS/2 und dem Warteschlangenpaket nicht interpretiert, sondern lediglich weitergegeben wird. Warteschlangen-Server und -klienten können diese Information zur Kommunikation einsetzen.

Das Warteschlangenpaket besitzt auch ein *Peek*-Leistungsmerkmal, das dem der benannten Pipes gleicht. Ein Unterschied besteht allerdings: Wenn ein Prozeß die in einer benannten Pipe vorliegenden Daten prüft und diese später liest, kann er sicher sein, daß die gelesene Meldung auch die geprüfte ist (da benannte Pipes immer nach dem Prinzip des FIFO arbeiten). Warteschlangen hingegen erlauben das Lesen von Meldungen in einer beliebigen Reihenfolge. Falls eine Warteschlange nach der Reihenfolge der Priorität gelesen wird, kann ein Prozeß eine Meldung prüfen. Wenn derselbe Prozeß einen Lesevorgang ausführt, kann nun eine andere Meldung die höchste Priorität besitzen und auf diese Weise am Anfang der Liste erscheinen. Um dieses Problem zu beheben, wird bei der Prüfung einer Meldung eine Nummer an den aufrufenden Prozeß übergeben. In einem nachfolgenden *DosReadQueue*-Aufruf kann die Nummer eingesetzt werden, um sicherzustellen, daß auch tatsächlich die geprüfte Meldung gelesen wird. In diesem Fall wird die normale Abarbeitung der Warteschlange ignoriert. Die Nummer kann auch in einem *DosPeekQueue*-Aufruf vergeben werden, um eine zweite oder nachfolgende Meldung in der Warteschlange zu untersuchen.

Ein weiterer wichtiger Unterschied zwischen Warteschlangen und benannten Pipes besteht darin, daß benannte Pipes Daten übertragen (das heißt, kopieren) und zwar vom Klienten- zum Serverprozeß. Bei Warteschlangen werden lediglich die Adressen der Daten übertragen, das Warteschlangenpaket kommt mit den Daten selbst nicht in Berührung. Die Daten, auf die sich die Warteschlangenmeldung bezieht, müssen also sowohl für den Klienten- als auch für den Serverprozeß adressierbar sein. Dies ist einfach, wenn sowohl Klienten- als auch Server-Threads zum selben Prozeß gehören. Falls die Threads von unterschied-

lichen Prozessen stammen, müssen die Daten, auf die sich die Warteschlangenmeldung bezieht, in einem gemeinsam verwendeten Speichersegment vorliegen.

Ähnliche Überlegungen gelten für die Wiederverwendbarkeit von Puffern. Eine Anwendung kann einen Speicherbereich sofort wiederverwenden, nachdem der Thread vom Schreiben der Daten in die benannte Pipe von der Ausführung zurückgekehrt ist. Bei der Benutzung einer Warteschlange darf der Klientenprozeß den Meldungsbereich nicht überschreiben, bis sichergestellt ist, daß der Server-Prozeß die Meldung erhalten hat.

Die Lösung der beiden Probleme - gemeinsam verwendeter Speicherbereich und dessen Wiederverwendung - ist der Einsatz des Speicherunterbelegungspaketes. Sowohl der Klient als auch der Warteschlangen-Server müssen gemeinsamen Zugriff auf ein Speichersegment besitzen, das vom Speicherunterbelegungspaket verwaltet wird. Der Klient belegt ein Speicherobjekt und schreibt die Meldung in die Warteschlange. Der Warteschlangen-Server kann die Warteschlangenmeldung adressieren, da diese in einem gemeinsamen ansprechbaren Speichersegment vorliegt. Ist die Übertragung der Meldung beendet, wird die Speicherunterbelegung zur Freigabe des Speicherobjektes aufgerufen. Der Klient muß sich nicht darum kümmern, ob die Bearbeitung der Meldung erfolgt ist, da der Klient für jede Meldung einen neuen Meldungspuffer belegt. Es wird davon ausgegangen, daß der Server die Meldung schnell genug abarbeitet, damit die Speicherunterbelegung stets genügend freie Speicherkapazität zur Verfügung hat.

Eine ähnliche Methode auf der Segmentebene stellt die Verwendung von Übergabespeicher dar. Der Klient belegt ein Übergabespeichersegment für jeden Meldungsinhalt, erzeugt die Meldung und übergibt dem Serverprozeß die Zugriffsmöglichkeit auf das Segment. Anschließend wird die Meldung (genauer die Adresse der Meldung) in die Warteschlange geschrieben. Beachten Sie, daß der Sender den Selektor des Empfängers als Datenadresse verwendet (nicht den eigenen Selektor). Wenn der Thread vom *DosWriteQueue*-Aufruf zurückkehrt, gibt der

Klient mit *DosFreeSeg* den Zugriff auf das Segment auf. Nach der Bearbeitung der Meldung durch den Serverprozeß wird auch von ihm das Speichersegment freigegeben. Da der *Queue*-Server der letzte Prozeß ist, der Zugriff auf das Segment hatte, wird das Segment wieder allgemein verfügbar.

Softwareentwickler sollten die Vor- und Nachteile von Warteschlangen, benannten Pipes und anderen Formen der IPC sorgfältig gegeneinander abwägen. Warteschlangen sind potentiell sehr schnell, da lediglich die Adresse der Daten und nicht die Daten selbst kopiert werden. Die erforderlichen Verwaltungs- und Wiederverwendungsoperationen für gemeinsam verwendete Speicherbereiche können die Zeiteinsparung bei geringen Übertragungsvolumina aber übertreffen. Allgemein kann gesagt werden, daß kurze Meldungen, die in der Reihenfolge des FIFO gelesen werden, über benannte Pipes übertragen werden sollten. Gleiches gilt für Anwendungen, die mit Klienten und Servern über ein Netzwerk kommunizieren. Große Datenmengen werden besser über Warteschlangen bearbeitet.

11.5 Dynamischer Datenaustausch DDE

DDE (*dynamic data exchange*) ist eine Form des IPC, die bei der Verwendung des Presentation Manager verfügbar ist. Die Schnittstelle des Presentation Manager ist meldungsorientiert, das heißt, das primäre Kommunikationsmittel zwischen einem Prozeß und dem Presentation Manager ist die Übergabe von Meldungen. Die Meldungsschnittstelle des Presentation Manager ermöglicht Anwendungen die Definition von Meldungen, die auf dem PC eine einzigartige Bedeutung besitzen. Beim DDE handelt es sich (streng betrachtet) um ein Protokoll, das neue Meldungen zur Kommunikation zwischen Anwendungsprogrammen definiert.

DDE-Meldungen können an ein bestimmtes Zielelement oder im Rundspruchverfahren an alle Anwendungen gesendet werden, die (auf einem bestimmten PC) den Presentation Manager verwenden. Üblicherweise gibt ein Klientenprozeß eine Rundspruchmeldung mit folgenden Inhalten aus: *Ist diese Informa-*

tion vorhanden? oder *Wird diese Serviceleistung bereitgestellt?* Falls keine Antwort übermittelt wird, ist dies gleichbedeutend mit der Antwort *Nein*. Eine Antwort auf derartige Rundspruchmeldungen enthält einen Identifikationscode (*Fensternummer*), die die Kommunikation zwischen den beiden Prozessen ermöglicht.

Der Rundspruchmechanismus und die Meldungsorientierung von DDE bieten in einer Multiprocessing-Umgebung ein hohes Maß an Flexibilität. So kann beispielsweise eine Anwendung Aktienkurse untersuchen, die über eine Schnittstelle verfügbar werden. Unter Verwendung des DDE kann eine Kalkulationstabelle der prüfenden Anwendung mitteilen, daß nur die Änderung bestimmter Aktienwerte (die in der Kalkulationstabelle berücksichtigt sind) weitergegeben werden. Eine andere Anwendung, man nenne sie hier *Marktalarm*, könnte das prüfende Programm veranlassen, eine Meldung an das Alarmprogramm auszugeben, wenn bestimmte Aktienkurse zu tief fallen oder sehr hoch über die normalen Kurswerte hinausreichen. DDEs können nur zur Kommunikation zwischen Anwendungen eingesetzt werden, die den Presentation Manager verwenden.

11.6 Signale

Signale sind asynchrone Merkmale, die Beachtung verdienen und wie Hardwareinterrupts arbeiten. Erreicht ein Signal (wie ein Hardwareinterrupt) einen Prozeß, unterbricht der Thread 1 des Prozesses nach der Abarbeitung der gerade ausgeführten Instruktion und veranlaßt die Ausführung des Programmteils an einer speziellen Adresse. Die in die Überlegung einzubeziehenden Punkte bei der Verwendung von Signalen werden in Kapitel 12 erläutert. Dieser Abschnitt dient lediglich der Erläuterung bei der Verwendung zur Kommunikation zwischen Prozessen.

Prozesse empfangen Signale üblicherweise als Antwort auf externe Ereignisse, auf die sofort reagiert werden muß. Beispiele für solche Ereignisse sind die Betätigung der Tastenkombination Ctrl-C durch den Benutzer oder ein vorzeitig beendeter Prozeß. Drei Signale (Flaggen A, B und C) werden aber von anderen

Prozessen veranlaßt, die eine explizite *DosFlagProcess*-API verwenden. *DosFlagProcess* ist eine einzigartige Form der IPC, da dieser Vorgang asynchron ist. Das empfangende Element muß keine Blockierung und keinen Polling-Wartevorgang auf ein Ereignis ausführen. Das Ereignis wird festgestellt, sobald der Scheduler CPU-Zeit an das Zielelement übergibt (es wird die entsprechende signalbearbeitende Routine ausgeführt).

DosFlagProcess hat einige Schwächen. Zum ersten überträgt das Signal nur wenig Informationen: lediglich die Nummer des Signals und ein einzelnes Argumentwort. Zum zweiten können Signale mit einigen Systemaufrufen in Konflikt geraten und drittens sieht OS/2 Signale mehr als Ereignisse denn als Meldungen an. Falls Signale schneller gesendet werden, als das Zielelement diese bearbeiten kann, unterschlägt OS/2 einige der sich ansammelnden Signale. Die Nachteile (Erläuterungen folgen in Kapitel 12) beschränken die Verwendung von Signalen auf Ausnahmefälle bei der Interprozeßkommunikation.

11.7 Verknüpfung von IPC-Formen

In den vorangehenden Abschnitten wurden alle Formen der IPC mit ihren Schwächen und Stärken erläutert. Oftmals bietet sich eine Verknüpfung der einzelnen IPC-Möglichkeiten an. So kann ein Prozeß beispielsweise eine benannte Pipe oder den DDE-Mechanismus zur Kommunikation mit einem anderen Prozeß verwenden und anschließend umfangreiche Datenmengen über gemeinsam verwendete Speicherbereiche übertragen. Eine Anwendung, die eine IPC-Schnittstelle mit sich bringt, sollte auch ein Dynlink-Paket zum Verbergen der Details des IPC enthalten. Dies erlaubt dem Entwickler die Verbesserung der IPC-Komponenten in künftigen Versionen, während die Schnittstellenkompatibilität mit den Klienten erhalten bleibt.

Kapitel 12

Signale

Die in OS/2 realisierten Methoden des Signalaustausches ähneln denen von Unix, sind aber mit diesen nicht identisch. Ein Signal entspricht in etwa einem Hardwareinterrupt, stammt aber von der Software. Ein Hardwareinterrupt sorgt dafür, daß die Register CS und IP sowie die Flaggenregister auf den Stapel gesichert werden und die Ausführung einer interruptbehandelnden Routine veranlaßt wird. Auch ein Signal sichert die CS-, IP- und Flaggenregister eines Anwendungsprogramms auf den Stapel und veranlaßt die Ausführung einer signalbehandelnden Routine. Die IRET-Anweisung gibt die Kontrolle in beiden Fällen an die Unterbrechungsadresse zurück. Signale unterscheiden sich von Hardwareinterrupts insofern, als daß sie Softwarekomponenten darstellen und keine Privilegänderungen, Stapelumschaltungen oder Programmcode für *Ring 0* umfassen.

OS/2 unterstützt sechs Signale - drei Signale mit festgelegter Funktion (Ctrl-C, Ctrl-Break und Programmbeendigung) und drei Signale zur allgemeinen Verwendung. Die Signale Ctrl-C und Ctrl-Break treten im Zusammenhang mit der Tastatur auf, das Signal zur Progammbeendigung entsteht bei der Beendigung eines Prozesses über einen *DosKill*-Aufruf.

Hinweis: Die signalbearbeitende Routine zur Prozeßbeendigung wird nicht in allen Fällen ausgeführt, in denen eine Prozeßbeendigung auftritt. Der Aufruf erfolgt nur als Reaktion auf einen DosKill-Aufruf. Normale Programmbeendigungen, GP-Fehler und ähnliche Ursachen aktivieren die signalbearbeitende Routine nicht.

Die drei allgemein verwendbaren Signale werden durch den Aufruf durch einen Thread (typischerweise ein Thread eines anderen Prozesses) erzeugt. Die signalbearbeitende Routine liegt in Form einer *far*-Unterroutine vor, das bedeutet, daß die Un-

terroutine die Kontrolle mit einer *far return*-Anweisung zurückgibt. Beim Ankommen eines Signals wird Thread 1 des Prozesses bei seiner momentanen Aufgabe unterbrochen und führt den Aufruf der signalbehandelnden Prozedur mit einem vom Signalgenerator bereitgestellten Argument aus. Falls die signalbearbeitende Routine die Kontrolle zurückgibt, führt die CPU den ablaufenden Prozeß an der Unterbrechungsstelle fort. Alternativ kann die signalbearbeitende Routine den Stapel löschen und zu einem anderen Punkt des Prozesses springen, von dem ab das Programm weiterbearbeitet wird (entspricht dem *longjmp* in der Programmiersprache C).

Hinweis: OS/2 arbeitet mit einem Trick, so daß die signalbearbeitende Routine die IRET-Instruktion (die Compiler üblicherweise nicht erzeugen) nicht selbst ausführen muß. Beim Einsprung in die signalbearbeitende Routine wird ein Adreßpunkt auf einen OS/2-Programmteil zurückgegeben, der eine IRET-Anweisung enthält.

Die Gemeinsamkeiten von Signalen und Hardwareinterrupts gehen noch weiter. Wie im Falle der Hardwareinterrupts blockiert das System weitere Signale aus derselben Quelle, so daß die Signalroutine nicht willkürlich nochmals aufgerufen werden kann. Die Signalroutine muß eine Sonderform des *DosSetSignalHandler*-Aufrufs ausgeben, um die Bearbeitung weiterer derartiger Signale zu veranlassen. Der Aufruf wird üblicherweise am Ende der signalbearbeitenden Routine ausgeführt, sofern die Signalroutine nicht gleichzeitig mehrfach verwendbar ist. Der *DosHoldSignal*-Aufruf wird ähnlich wie bei Hardwareinterrupts die CLI-Anweisung verwendet, um die Abarbeitung kritischer Bereiche vor der Unterbrechung durch ein Signal zu schützen.

Im Gegensatz zu Hardwareinterrupts besitzen Signale keine Interruptpriorität. Wenn ein Signal auftritt, dessen Signalroutine derzeit verfügbar ist, wird in die Signalroutine gesprungen (auch wenn dafür eine andere Signalroutine unterbrochen werden muß). Ankommende Signale, die in der Bearbeitungszeit eines früheren gleichartigen Signals ankommen, werden aufgehalten, bis das vorherige Signal abgearbeitet wurde. Wie bei Hardwareinterrupts gibt es lediglich eine Ja-/Nein-Flagge (keine Fehlerspezifikation). Falls drei Signale einer Art während der

Bearbeitung eines Signals ankommen, tritt nur ein Signalereignis auf, da die entsprechende signalbearbeitende Routine belegt ist.

Ein Signalereignis tritt im Zusammenhang mit einem Prozeß auf, dessen Ausführungs-Thread für die Abarbeitung einer Signalroutine unterbrochen wird. Das Signal veranlaßt nicht die Beendigung der CPU-Zuweisung an andere Prozesse, um die signalbearbeitende Routine des ersten Prozesses auszuführen. Wenn OS/2 ein Signal für einen Prozeß erhält, wird lediglich ein Zeichen gesetzt, das aussagt: *Bei der nächsten Abarbeitungszeit sollen die CS-, IP- und Flaggenwerte auf den Stapel gesichert und der Sprung zu einer bestimmten Programmstelle veranlaßt werden.* Das System verwendet weiterhin die normalen Prioritätsregeln zur Zuweisung der CPU-Zeit an einzelne Threads. Der Dispatcher-Code, der die CPU-Zeiten zuteilt, liest das Signalzeichen und führt die entsprechenden Operationen aus, wenn die Zeitverwaltung (der Scheduler) dem Thread die nächste "Zeitscheibe" zuteilt.

Da ein Signal (*Pseudo-Interrupt*) im Grunde lediglich ein Trick des Dispatchers ist, werden Signalroutinen nicht in Ring 0 (wie Hardwareinterruptroutinen) ausgeführt, sie laufen wie alle anderen Anwendungs-Threads in Ring 3 ab. Soweit OS/2 betroffen ist, hat der Prozeß keinen besonderen Status bei der Ausführung einer Signalroutine. Es gibt auch keine Sonderregeln für die Verhaltensweisen eines Threads in einer Signalroutine.

Der Signalempfang durch Thread 1 bei der Bearbeitung eines Anwendungsprogramms oder einer Dynlink-Routine ist einfach: Das System speichert die CS-, IP- und Flaggenwerte und die signalbearbeitende Routine speichert die restlichen benötigten Informationen. Der vollständige Registersatz kann nach der Bearbeitung des Signals wieder geladen werden und Thread 1 nimmt die normale Programmausführung wieder auf. Falls Thread 1 einen Systemaufruf ausführt und dabei im Kernel arbeitet, verkompliziert sich der Vorgang. Ein Interrupt kann im Ring-0-Code nicht ablaufen und OS/2 kann nicht riskieren, daß die Signalroutine fehlerhaft ist und nicht zurückkehrt, wobei die internen Werte von OS/2 in einem unbestimmten Status zurückgelassen würden. Eine signalbearbeitende Routine kann den Stapel des Thread 1 löschen, die Signalbearbeitung abbrechen und zu

einem anderen Teil der Anwendung springen. Eine Rückkehr von signalbearbeitenden Aufrufen erfolgt in diesem Falle nicht. Beispielsweise kann eine Anwendung in eine *Eingabeaufforderung anzeigen und Befehl ausführen*-Schleife springen, wenn der Benutzer die Tastenkombination Ctrl-C betätigt hat. Ein Signal wird bei der Ankunft während eines Systemaufrufs vorgemerkt und Thread 1 führt den Ring-0-Code weiterhin aus. Das System vervollständigt die Operation vor der Erkennung des Signals oder bricht die Operation ab und erkennt anschließend das Signal. Falls die Operation schnell ausgeführt werden soll, wird sie vom System beendet und das Signal entdeckt, wenn die CPU die Ausführung des Ring-3-Codes wieder aufnehmen will.

Alle Operationen, die keine Ein- oder Ausgabe darstellen, werden als schnell angesehen. Eine Ausnahme bilden explizite Blockieroperationen wie beispielsweise *DosSleep* und *DosSemWait*. Ein-/Ausgabeoperationen hängen von der angesprochenen Einheit ab. Die Diskettenein-/-ausgabe kann schnell beendet werden, Tastatureingaben und serielle Ein-/Ausgaben hingegen im allgemeinen nicht. Es kann nicht abgeschätzt werden, wie lange ein Benutzer zur Eingabe einer Befehlszeile oder Antwort benötigt. Bei der Verwendung langsamer Einheiten oder bei der Abarbeitung von Operationen mit unvorhersehbarer Länge beendet OS/2 oder der Einheitentreiber die Operation vorzeitig und gibt die Kontrolle mit einem Fehlercode an die Anwendung zurück. Das Signal wird erkannt, wenn die CPU den Ring-3-Anwendungscode weiter abarbeiten will, der nach dem unterbrochenen Systemaufruf folgen würde.

Obgleich dem Anwendungsprogramm ein Fehlercode übergeben wird, der die Unterbrechnung des Systemaufrufs anzeigt, kann es vorkommen, daß die Anwendung den Systemaufruf zur Beendigung der Aufgabe nicht nochmals ausführen kann. Im Fall der Einheitenein- oder -ausgabe kann nicht mit Bestimmtheit festgestellt werden, wieviel Daten des Gesamtdatenbestandes geschrieben oder gelesen wurden. Falls eine Ausgabeoperation nicht nochmals ausgeführt wird, kann ein Teil der Daten beim Schreibvorgang verloren gegangen sein. Bei der wiederholten Ausführung einer Ausgabeoperation können Daten am Anfang

einer Datenmenge doppelt übertragen werden. Im Falle eines *DosSleep*-Aufrufs kann das Anwendungsprogramm nicht feststellen, wieviel der angeforderten Wartezeit bereits verstrichen ist. Dies stellt kein Problem dar; üblicherweise wird die Tastatureingabe unterbrochen. Im Falle der häufig vorkommenden Signale (Ctrl-C, Ctrl-Break und Prozeßbeendigung) löscht die Anwendung die bereits teilweise vorliegende Tastatureingabe. Bei Anwendungen, die mit relativ langsamen Einheiten arbeiten oder IPC-Flaggensignale einsetzen, ist dies zu berücksichtigen.

Ein Prozeß kann mehrere Threads besitzen, aber nur Thread 1 wird zur Ausführung der Signalroutine eingesetzt. Aus diesem Grund sollte ein Prozeß nicht Thread 1 beenden und andere Threads weiter ausführen. Ein solcher Prozeß kann keine Signale mehr empfangen. Dies führt zu einer offensichtlichen Lösung des Problems der unterbrochenen Systemaufrufe: Anwendungen, die nichtunterbrechbare Systemaufrufe ausführen, sollten andere Threads als Thread 1 für diese Aufgabe einsetzen. Im denkbar schlechtesten Fall kann Thread 1 die Aufgabe des Wartens auf Signale übertragen werden: Dazu wird eine RAM-Semaphore blockiert (die nicht freigegeben wird) oder eine *DosSleep*-Schleife ausgeführt.

Es gibt noch einige wichtige Details im Zusammenhang mit Signalen, die im folgenden erläutert werden sollen. Zunächst einmal muß sich der Benutzer einer hochentwickelten Sprache wie C nicht über die Speicherung der Registerinhalte in der signalbearbeitenden Routine sorgen. Die Laufzeitpakete der Hochsprachen enthalten üblicherweise Programmcode für diese Details. Soweit das Anwendungsprogramm betroffen ist, wird die Signalroutine mit einem asynchronen *far*-Aufruf ausgeführt und kann von der Signalbearbeitung durch eine *return*-Anweisung zurückkehren. Es können nur Signale empfangen werden, wenn die Anwendung zuvor eine Signalanforderung ausgeführt hat. Aus diesem Grund müssen keine Signalroutinen erstellt werden, falls eine Anwendung nicht explizit Signale abfragt. Ein Prozeß kann für jedes Signal nur eine signalbearbeitende Adresse besitzen. Allgemein verwendbare Dynlink-Routinen (die von Anwendungen aufgerufen werden können, die nicht

mit den Dynlink-Paketen bestückt sind) sollten keine signalbearbeitenden Adressen setzen. Dadurch kann eine vom Klientenprogramm gesetzte Routine überschrieben werden.

Signale verhalten sich zu kritischen Bereichen ähnlich wie Interrupts. Die Ankunft eines Signals bei der Bearbeitung eines kritischen Bereiches durch Thread 1 (der mit einer Semaphore geschützt ist) kann dazu führen, daß die Semaphore für den kritischen Bereich nicht freigegeben wird, wenn die signalbearbeitende Routine nicht zur Unterbrechungsstelle zurückkehrt. Selbst wenn die Signalroutine einen Rücksprung durchführt, kann ein *Deadlock* auftreten, wenn der Zugriff auf den kritischen Bereich im Rahmen der Signalbearbeitung versucht wird (vielleicht wird ein Dynlink-Paket aufgerufen, von dem nicht bekannt ist, daß dort ein kritischer Bereich enthalten ist). Dynlink-Pakete müssen dieses Problem über den *DosHoldSignal*-Aufruf bearbeiten, der den Anweisungen CLI/STI der Hardwareinterrupts entspricht: Der *DosHoldSignal*-Aufruf hält ankommende Signale vom Zugriff ab, bis die Bereiche freigegeben werden. Aufgehaltene Signale sollten in ein oder zwei Sekunden freigegeben werden, damit der Benutzer die Tastenkombination Ctrl-C nicht wiederholt anschlägt und annimmt, daß das System abgestürzt ist. Anwendungen können den *DosHoldSignal*-Aufruf einsetzen oder einfach sicherstellen, daß Thread 1 niemals einen kritischen Bereich abarbeitet (indem Thread 1 nur für die Signalbearbeitung eingesetzt wird).

Die Tastenkombinationen Ctrl-C und Ctrl-Break sind einheitenspezifische Operationen. Das Setzen einer Routine für die Signalbearbeitung ist eine Form der Ein-/Ausgabe an die Tastatureinheit. Daher sollten Anwendungsprogramme vor dem Zugriff sicherstellen, daß es sich bei der zugeordneten Einheit tatsächlich um eine Tastatur handelt (siehe Kapitel 14).

Kapitel 13

Presentation Manager und VIO

In den ersten Kapiteln dieses Buches wurde bereits die Bedeutung einer leistungsfähigen flexiblen graphischen Benutzerschnittstelle erläutert. Kaum angesprochen wurden allerdings die Probleme, die mit einer derartigen Schnittstelle zusammenhängen. Die Programmierung einer graphischen Bedieneroberfläche ist mit sehr viel Aufwand verbunden, der im allgemeinen für ein einziges Anwendungsprogramm nicht zu rechtfertigen ist. Für den Benutzer wiederum ist es schwierig, die graphischen Schnittstellen unterschiedlicher Anwendungen zu beherrschen. Das Ideal wäre eine graphische Standardoberfläche, die möglichst einfach mit Anwendungsprogrammen verknüpfbar ist. Hier kommt nun der *Presentation Manager* von OS/2 ins Spiel.

Der Presentation Manager ist als Kombination eines Dynlink-Subsystems und eines *Daemon*-Prozesses implementiert und stellt folgende Leistungsmerkmale bereit:

- Graphische Fenstertechniken mit hohem Datendurchsatz.
- Ein leistungsfähiges Benutzerschnittstellenmodell mit *Drop-Down*-Menüs, Laufbalken, Ikonen (Sinnbildern), Maus- und Tastaturschnittstellen. Die Verfügbarkeit der meisten Leistungsmerkmale ist für eine Anwendung nicht zwingend erforderlich, jedes Programm kann die Standardroutinen durch eigene Routinen ersetzen.

- Geräteunabhängigkeit. Der Presentation Manager enthält eine hochentwickelte Einheitenschnittstelle, so daß eine Optimierung des Datendurchsatzes als Aufgabe an eine Graphikkarte abgetreten werden kann.

Die Verwendung der Oberfläche des Presentation Manager ist in einer Anwendung nicht übermäßig schwierig, aber doch mit einem gewissen Aufwand verbunden. Daher bietet OS/2 neben dem Presentation Manager noch eine zweite Benutzerschnittstelle mit dem Namen VIO, die in erster Linie zeichenorientiert ist und der ROM-BIOS-Bildschirmschnittstelle von MS-DOS ähnelt. Die ursprüngliche Version von OS/2 enthält lediglich das VIO-Paket, das als separate Komponente implementiert wurde. Die nächste Version wird auch den Presentation Manager umfassen, so daß der Anwender zwischen VIO-Paket und Presentation Manager frei wählen kann.

Hinweis: VIO ist ein Ausdruck, der drei Dynlink-Subsysteme umfaßt: KBD (für Keyboard), VIO (für Video I/O; Anzeigeadapter) und MOU (für Mouse).

Presentation Manager und VIO stellen im Prinzip Äquivalente zu Einheitentreibern dar. Sie sind als Dynlink-Pakete implementiert, da sie einheitenabhängig sind und bei der Verwendung unterschiedlicher Geräteeinheiten ersetzt werden müssen. Statt echter Einheitentreiber werden dynamische Verknüpfungen verwendet, da sie den hohen Datendurchsatz zu Bildschirmeinheiten sicherstellen. Ein einfacher, direkter Aufruf des Programmcodes stellt die einzelnen Pixel auf dem Bildschirm dar. Die Dynlink-Schnittstelle ermöglicht es, daß der Presentation Manager teilweise als Dynlink-Subsystem und teilweise als Daemon-Prozeß, auf den das Subsystem zugreift, implementiert ist. Die Möglichkeit der Substitution eines Dynlink-Paketes gegen einen Einheitentreiber (vollständig oder teilweise) stellt ein wichtiges Leistungsmerkmal von OS/2 dar. Es hängt von den Umständen ab, ob ein Einheitentreiber oder ein Dynlink-Paket besser für den Einsatz mit einem Anwendungsprogramm geeignet ist. Die Entwicklung eines Einheitentreibers ist einfacher, dies gilt insbesondere, wenn ungültige Klientenaufrufe und Argumente, fehlerhafte Prozesse oder unerwartete vorzeitige Beendigungen nicht berücksichtigt werden sollen. Einheitentreiber

können auch Hardwareinterrupts empfangen. Auf der anderen Seite haben Dynlink-Einheitenschnittstellen eine hohe Performance, wenn eine Anwendung häufig Aufrufe an eine Einheit wie den Bildschirm ausgibt.

Die VIO und der Presentation Manager sind in ihrer Struktur sehr komplex; die detaillierte Beschreibung würde den Rahmen des vorliegenden Buches sprengen. Daher sollen im folgenden nur die grundsätzlichen Eigenschaften erläutert werden.

Die VIO-Schnittstelle ist im wesentlichen zeichenorientiert. Graphikanwendungen werden unterstützt, aber nur soweit, daß die direkte Manipulation des Anzeigeadapters zur Umgehung der VIO-Schnittstelle und die Bereitstellung besonderer Schnittstellen zur Umschaltung zwischen Graphikanwendungen (siehe unten) erlaubt werden. Das Basis-VIO-Paket ähnelt der INT 10-/INT 16-Schnittstelle des ROM-BIOS unter MS-DOS. Im Grunde ist das VIO-Paket eine Erweiterung des ROM-BIOS. Real-Modus-Anwendungen, die auf dem Interrupt 10 basieren, können problemlos an die VIO-Schnittstelle angepaßt werden. Das VIO-Paket ist teilweise oder vollständig ersetzbar, damit Anwendungen, die derzeit mit der VIO-Schnittstelle arbeiten, später ohne größeren Programmieraufwand den Presentation Manager nutzen können.

13.1 PM oder VIO?

Der Presentation Manager ersetzt die VIO/KBD/MOU-Pakete. Die VIO-Schnittstelle und der Presentation Manager sind in ihrer Verwendung nicht immer klar voneinander trennbar. Häufig stellt sich die Frage, welche Schnittstelle man einsetzen soll. Die Standardschnittstelle für neue Anwendungen sollte immer der Presentation Manager sein. Das Fehlen des Presentation Manager in OS/2 Version 1.0 ist ausschließlich auf Termingründe zurückzuführen. Besitzer der Version 1.0 werden den Presentation Manager nachträglich erhalten, sobald er verfügbar ist. In allen künftigen OS/2-Versionen wird der Presentation Manager enthalten sein. Der Presentation Manager wird in allen wichtigen OS/2-Installationen auf Personal Computern zu finden sein,

so daß der potentielle Markt für ein Softwarepaket durch den Einsatz des Presentation Manager nicht eingeschränkt wird.

Der Presentation Manager bietet einer Anwendung eine leistungsfähige graphische Benutzerschnittstelle, die in ihrer Standardausführung unmittelbar zur Verfügung steht. Zugleich kann der Presentation Manager an eine bestimmte Anwendung angepaßt werden, so daß der Wert eines Anwendungsprogramms durch Einsatz der graphischen Benutzeroberfläche deutlich gesteigert wird.

Bei einem bereits fertigen Anwendungsprogramm für den Real-Modus stellt sich die Frage, ob der Aufwand für die PM-Anpassung gerechtfertigt ist. Eine Anwendung ist einfach zu portieren, um die VIO-Schnittstelle nutzen zu können. Solche Programme funktionieren auch unter dem Presentation Manager, sie vermeiden aber die Benutzung der Graphiken und die Interaktion mit dem Presentation Manager. Der Benutzer kann sich die Bildschirmdarstellung der VIO-Anwendung in einem Fenster anzeigen lassen, dabei wird jedoch keine Verknüpfung mit dem Anwendungsprogramm hergestellt. Bei der Entwicklung von Anwendungssoftware steht der Programmierer vor drei Wahlmöglichkeiten:

- Verwendung der VIO-Schnittstelle im Zeichenmodus. In einer Presentation Manager-Umgebung funktioniert dies und sollte für die Portierung von Real-Modus-Anwendungen eingesetzt werden. Die VIO-Schnittstelle wird auch von der Family API unterstützt. Dieser Modus ist mit den Leistungsmerkmalen der Family API kompatibel.
- Verwendung der VIO-Schnittstellenleistungsmerkmale zur Umgehung der VIO und der direkten Manipulation der Bildschirmdarstellung im Text- oder Graphikmodus. Auch diese Möglichkeit kann in einer Presentation Manager-Umgebung eingesetzt werden, die Anwendung kann aber nicht in einem Fenster ablaufen. Das erstellte Anwendungsprogramm kann bei sorgfältiger Programmierung mit der Family API kompatibel sein.

- Verwendung des Presentation Manager - einer hochentwikkelten, aber einfach zu benutzenden Schnittstelle. Der Presentation Manager stellt eine Möglichkeit zur Ausführung von Anwendungen bereit, die ein breiter Benutzerstandard werden wird. Dies kann aufgrund der Leistungsmerkmale der Schnittstelle sowie der Unterstützung durch wichtige Softwarehersteller und durch die enge Verknüpfung mit OS/2 mit Sicherheit gesagt werden. Der Benutzer hat einen Vorteil, sich nicht bei jeder neuen Anwendung mit einer anderen Schnittstelle vertraut machen zu müssen. Microsoft ist von der Bedeutung einer leistungsfähigen graphischen Benutzerschnittstelle überzeugt. In künftigen OS/2-Versionen wird der Presentation Manager zahlreiche Verbesserungen erfahren. Viele Verbesserungen werden zu bereits existierenden Presentation Manager-Applikationen hinzugefügt werden, andere werden die API-Schnittstelle erweitern. Die rudimentären Schnittstellen der PC-Software der ersten Generation sind heute nicht mehr wettbewerbsfähig. Ein Betriebssystem wie OS/2 kann die Gestaltung von modernen Bedieneroberflächen wesentlich vereinfachen.

Es folgt, daß der Presentation Manager eine weitgehend geeignete Benutzerschnittstelle für Anwendungssoftware ist. Die Presentation Manager-API wird auch weiterhin erweitert und verbessert. Für die nächste Zukunft werden auch INT-10-ähnliche VIO-Funktionen und der direkte Bildschirmzugriff über VIO unterstützt werden. Es muß aber klar gesagt werden, daß hier mit einem längerfristigen Ausbau nicht zu rechnen ist. In vielen Fällen wird es sich daher anbieten, Real-Modus-Anwendungsprogramme zunächst kurzfristig auf VIO zu portieren, um Zeit für die vollständige Überarbeitung für die Presentation Manager-Version zu gewinnen.

13.2 Hintergrundein-/-ausgabe

Ein Prozeß befindet sich im Hintergrund, wenn er mit dem Benutzer nicht in Interaktion treten kann. In einer Presentation Manager-Umgebung bedeutet dies, daß kein Fenster des Prozesses die Tastatur anspricht. Dennoch kann das entsprechende Fenster am Bildschirm sichtbar sein, beispielsweise als Ikone. In einer VIO-Umgebung befindet sich ein Prozeß im Hintergrund, wenn der Benutzer eine andere Bildschirmgruppe gewählt hat. In diesem Fall ist die Bildschirmanzeige der Anwendung nicht sichtbar.

Für eine Presentation Manager-Anwendung bestehen bei der Aktualisierung der Bildschirmanzeige im Hintergrund keine Probleme, die Anwendung kann den PM weiterhin zur Änderung des Fensterinhaltes aufrufen. Es besteht die Möglichkeit, eine Anwendung darüber zu informieren, ob sie im Hinter- oder Vordergrundmodus arbeitet. Eine Anwendung braucht auf diesen Status aber keine Rücksicht zu nehmen, sie kann einfach weiterarbeiten. Die Hintergrundein-/-ausgabe kann unabhängig davon vorgenommen werden, ob sie text- oder graphikorientiert ist.

VIO-Anwendung können im Hintergrund ebenfalls Ein-/Ausgaben ausführen. Das VIO-Paket enthält für jede Bildschirmgruppe einen logischen Bildschirmpuffer. Über VIO-Aufrufe kann die Aktualisierung der Bildschirmanzeige auch im Hintergrundmodus fortgeführt werden. Die Aktualisierungen werden im logischen Bildfschirmpuffer vorgenommen und dieser dann bei der Umschaltung in den Vordergrund in den physikalischen Bildschirmpuffer kopiert.

13.3 Graphik unter VIO

Das VIO-Paket arbeitet zeichenorientiert und stellt eine Reihe von Dienstleistungen für Textanwendungen bereit. Bei einer Änderung der Bildschirmzuweisung speichert das VIO-Paket die momentane Bildschirmdarstellung und lädt die der neuen Software. Viele Anwendungen im Real-Modus umgehen das ROM-

BIOS (zumindest teilweise), dies ist über die VIO-Schnittstelle unter OS/2 schwieriger. Wenn Sie die VIO-Schnittstelle zur Anzeige von Graphiken umgehen, gibt es zwei gravierende Nachteile:

- Die Anwendung ist einheitenabhängig, da direkt auf die Bildschirmhardware zugegriffen wird.
- Die VIO sichert und lädt den Bildschirminhalt bei Umschaltungen zwischen Anwendung nicht mehr automatisch. Das Programm muß eine besondere VIO-Schnittstelle zum Bereitstellen eigener Funktionen verwenden.

Die folgenden Erläuterungen zeigen, wie man mit Anwendungsprogrammen die VIO-Schnittstelle umgehen kann und so den direkten Zugriff auf den Bildschirm erlangt.

Der Zugriff auf die Bildschirmhardware ist einfach: Der VIO-Aufruf *VioGetBuf* stellt einen Selektor auf den Bildschirmpuffer bereit und gibt dem Ring-2-Code der Anwendung die Möglichkeit, die Register des Bildschirm-Controllers zu verändern. Probleme treten durch die Bildschirmumschaltungsmerkmale von OS/2 auf. Wenn der Benutzer die Anwendung in eine Hintergrundbildschirmgruppe umleitet, ist der Bildschirmspeicherinhalt einer anderen Software zugeteilt. Der Bildschirmspeicher der Anwendung wird im RAM temporär abgelegt. Falls ein Programm diesem Prozeß keine Beachtung schenkt und versehentlich den Bildschirmspeicher aktualisiert oder auf den Bildschirm-Controller zugreift, während dieser einer anderen Bildschirmgruppe zugeordnet ist, kommt es zu Zugriffskonflikten.

Zwei wichtige Aspekte sind bei einer Bildschirmumschaltung zu beachten: Wie erkennt ein Programm, ob es im Hintergrundmodus abläuft? Wer sichert und speichert die Bildschirmdarstellung, an welcher Stelle geschieht dies?

Der *VioScrLock*-Aufruf dient dem Bildschirmzugriff. Vor jedem Zugriff auf den Anzeigespeicher oder -Controller muß eine Anwendung den *VioScrLock*-Aufruf ausführen. Während des Aufrufes kann OS/2 keine Bildschirmumschaltung ausführen. Das Anwendungsprogramm kann seine Aufgabe erledigen und

das "Sperren" des Bildschirms wieder aufheben. Unterbleibt dies in einem gewissen Zeitrahmen, "hängt" das System. Dies gilt nicht nur für explizite Bildschirmumschaltbefehle, die der Benutzer einsetzt, sondern auch für andere Aktionen, die zur Bildschirmumschaltung führen (beispielsweise für fehlerbearbeitende Routinen aufgrund von Hardwarefehlern). Hardwarefehler können dem Benutzer mitgeteilt werden, während die Bildschirmsperre aktiv ist. Falls OS/2 die Bildschirmdarstellung umschalten muß, eine nicht korrekt ablaufende Anwendung aber eine Bildschirmsperre gesetzt hat, beendet OS/2 die Sperre und führt die Umschaltung nach einer vorgegebenen Zeitspanne (üblicherweise einige Sekunden) aus. Die Lösung ist noch nicht zufriedenstellend, da der Vorgang für das manipulierte Anwendungsprogramm üblicherweise mit einer zerstörten Bildschirmdarstellung endet. Die Bildschirmsperre und das Aufheben der Sperre werden relativ schnell ausgeführt. Aus diesem Grund können sie häufig zum Schutz der aktuellen Schreiboperationen in die Bildschirmdarstellung verwendet werden. Eine Sperrung sollte aber nicht aktiv bleiben, während ein Anwendungsprogramm weitere Berechnungen oder Aufgaben ausführt. Grundsätzlich sollte eine Anwendung den *VioScrLock*-Aufruf zum Schutz eines Ein-/Ausgabeblockes einsetzen, der in seiner Gesamtheit zum Schreiben bereitsteht und keine weiteren Berechnungen erfordert. Beispiele von blockorientierten Operationen sind das Rollen der Bildschirmdarstellung, das Löschen der Bildschirmdarstellung oder das Schreiben einer Zelle in einem Kalkulationstabellenprogramm.

VioScrLock muß sowohl zum Schutz der Codesequenzen, die den Bildschirm programmieren, als auch zum Schutz von Codesequenzen, die in den Bildschirmspeicher schreiben, eingesetzt werden. Befehlsfolgen zur Programmierung der Peripherieeinheiten können nicht unterbrochen werden. So kann beispielsweise eine Befehlsfolge, in der im ersten Schritt ein Multiplex-Register ausgewählt wird und im zweiten Schritt das ausgewählte Register modifiziert wird, nicht unterbrochen werden, da der erste Schreibvorgang die Peripherieeinheit in einen Sonderstatus

versetzt. Diese Befehlsfolgen müssen durch ein *lock/unlock*-Paar geschützt werden.

Beim *VioScrLock*-Aufruf eines Anwendungsprogramms kann der Fehlercode *Bildschirm ist nicht verfügbar* zurückgegeben werden. Dies bedeutet, daß der Bildschirm für diese Anwendung in den Hintergrund geschaltet wurde - das Programm darf die Darstellung nicht verändern. Im allgemeinen wird die Bildschirmdarstellung vom VIO-/Presentation Manager-Paket in einem Puffer abgelegt. Da das Programm den Speicherbereich und das verwendete Format zur Ablage der Bildschirmdarstellung nicht kennt, muß es warten, bis seine Bildschirmgruppe wieder in den Vordergrund geschaltet wird. Üblicherweise wird eine Form des *VioScrLock*-Aufrufs verwendet, der den Thread beschäftigt hält, bis sich die entsprechende Bildschirmdarstellung wieder im Vordergrund befindet und die Bildschirmpuffer die Bildschirmdarstellung für diesen Prozeß enthalten.

Es gibt Anwendungen, für die dieser einfache Mechanismus nicht ausreichend ist. Häufig sollen Anwendungen im Hintergrundmodus die später im Programm erfolgende Bildschirmdarstellung weiterhin aktualisieren. Oder es ist für eine Anwendung einfacher, eine Bildschirmausgabe neu zu erstellen, statt den alten Bildschirminhalt zu verändern. Ein weiterer schwieriger Fall ist die Verwendung EGA-kompatibler Hardware im EGA-Modus. Da der Standard-EGA viele Nur-Schreib-Register enthält, können Elemente wie das VIO-Paket eine EGA-Bildschirmdarstellung nicht korrekt speichern und wieder laden (da wichtige Informationen nicht gelesen werden können). Aus diesem Grunde unterstützen sowohl die VIO als auch der Presentation Manager eine Schnittstelle, die Anwendungen das Speichern und Laden eigener Bildschirmdarstellungen erlaubt.

Wie erkennt das System den dazu notwendigen Prozeß? Prozesse können das System in vielfältiger Weise aufrufen, das System kann aber keine Prozesse aufrufen. OS/2 behandelt diese Situation durch die Invertierung der üblichen Bedeutung der Begriffe *Aufruf* und *Rückgabe*. Bei der ersten Aktualisierung einer Bildschirmdarstellung durch einen Prozeß wird ein Extra-

Thread angelegt und mit diesem die *VioSavRedrawWait*-Funktion aufgerufen. Der Thread kehrt von dem Aufruf nicht sofort zurück. Vielmehr hält das VIO-Paket den Thread, bis die Zeit für eine Bildschirmumschaltung gekommen ist. Um dem Prozeß mitzuteilen, daß die aktuelle Bildschirmdarstellung nun gespeichert werden muß, entläßt VIO den Thread und dieser kehrt vom *VioSavRedrawWait*-Aufruf zurück. Der Prozeß speichert den Anzeigestatus und den Bildschirminhalt (üblicherweise unter Verwendung des Threads, mit dem der Aufruf vorgenommen wurde). Nach Beendigung der Speicheroperation wird wiederum ein *VioSavRedrawWait*-Aufruf ausgeführt. Das VIO-Paket erkennt, daß die Speicherung abgeschlossen ist und der Bildschirm nun umgeschaltet werden kann. Der Zyklus wird zurückgesetzt, so daß dem Prozeß wiederum mitgeteilt werden kann, wann die Zeit zur Speicherung der Bildschirmdarstellung gekommen ist. Dies macht die Rückkehr des *VioSavRedrawWait*-Aufrufes zu einem "*System-zum-Prozeß-Aufruf*". Der erneute Aufruf von *VioSavRedrawWait* entspricht der Rückkehr vom Prozeß zum System.

Das Konzept von OS/2 wurde darauf ausgelegt, daß das System möglichst keine Prozesse zur Unterstützung einer Systemaktivität (wie dem Umschalten der Bildschirmdarstellung) aufruft. Der Hintergrund: Kein Prozeß unter OS/2 soll in der Lage sein, das System zum Absturz zu bringen. Das VIO-Paket hält die Umschaltung zurück, bis der Prozeß die Bildschirmdarstellung gespeichert hat. Was passiert aber, wenn der Prozeß erfolglos endet und die Speicherung nicht vollständig ausgeführt werden konnte? Der Bildschirm ist in einem unbekannten Status und kein Prozeß kann dann Bildschirm oder Tastatur verwenden. Aus Benutzersicht handelt es sich um einen Systemabsturz. Obgleich andere Prozesse im System weiterhin funktionieren (können), hilft dies dem Benutzer wenig, da er die Arbeit dieser Prozesse nicht mehr speichern kann.

Die Entwickler von OS/2 befanden sich hier in einer schwierigen Situation: Anwendungen sollten ihre Bildschirmdarstellungen selbst speichern können, wenn sie die erweiterten Modi des EGA einsetzen. Dieses Leistungsmerkmal verletzt aber die Ziel-

vorgabe des Systemkonzeptes, daß kein Absturz möglich sein darf. Da der EGA unterstützt werden mußte und das System nicht absturzgefährdet sein durfte, wurde eine zweiteilige Lösung gewählt.

Der erste Teil der Lösung betrifft einen der häufigsten Gründe für einen Prozeßabsturz: Hardwarefehler. Beim Auftreten eines Hardwarefehlers verwendet der Hardwarefehler-*Daemon* den Bildschirmumschaltmechanismus zur Übernahme der Bildschirm- und Tastaturkontrolle. Der Hardwarefehler-Daemon speichert die existierende Bildschirmdarstellung und verhindert, daß die Vordergrundanwendung die Bildschirm- und/oder Tastaturkontrolle erlangt. Dennoch könnte folgende Situation auftreten: Der Hardwarefehler-*Daemon* verwendet den Bildschirmumschaltmechanismus, der dem Vordergrundprozeß die Speicherung der eigenen Bildschirmabbildung erlaubt. Spricht der Prozeß während der Speicherung die Einheit mit dem Hardwarefehler an, kann es zu einem *Deadlock* kommen. Die Einheit, in der der Fehler aufgetreten ist, wird keine Anforderung mehr bedienen, bis der Hardwarefehler beseitigt ist. Der Hardwarefehler kann nicht beseitigt werden, bis der Daemon die Kontrolle übernimmt. Der Daemon-Prozeß kann die Kontrolle aber nicht übernehmen, bis der Vordergrundprozeß die Speicherung der Bildschirmdarstellung beendet hat, und der Vordergrundprozeß kann die Speicherung nicht vervollständigen, bis die Einheit seine Anforderung bearbeitet. Beachten Sie, daß zum Auftreten dieses Deadlocks keine explizite Ein-/Ausgabeoperation des Vordergrundprozesses nötig ist. Die einfache Belegung von Speicher oder die Bezugnahme auf ein Segment können zu einer Speicherauslagerung oder zum Laden von Speicherbereichen führen, wobei das Gerät angesprochen wird, in dem der Fehler aufgetreten ist.

Die Lösung bestand darin, die Bildschirmumschaltung bei einem Hardwarefehler nur teilweise als Umschaltung zu gestalten. Es wird zuerst nur ein Teil des Bildschirmspeichers (üblicherweise vier Kbyte) abgelegt - gerade genug, damit der Hardwarefehler-Daemon einen Teil der Bildschirmdarstellung mit Text füllen kann. Wieviel RAM die Anwendung zur Speicherung einer

Bildschirmdarstellung benötigt, ist hierbei nicht von Belang, da in den Textmodus umgeschaltet wird und der Hardwarefehler-Daemon nur einen kleinen Teil der Bildschirmanzeige überschreibt. Dies bedeutet aber, daß der Hardwarefehler-Daemon nicht in eine dritte Bildschirmgruppe umschalten kann, da die Darstellung nicht vollständig gesichert wurde.

Das VIO-Paket und der Hardwarefehler-Daemon halten immer genug freie Kapazität bereit, so daß eine Fehlermeldung ohne Speicherauslagerung angezeigt werden kann. Nach der Beendigung der Anzeige wird die ursprüngliche Bildschirmdarstellung wieder geladen. Es wurde bereits erläutert, daß das VIO-Paket den Status des Bildschirm-Controllers nicht beeinflussen kann, dies ist nur Anwendungsprogrammen möglich. Die *VioModeWait*-Funktion wird verwendet, um der Anwendung mitzuteilen, daß der Status wiederhergestellt werden muß.

Alle Prozesse, die einen EGA oder einen ähnlichen Controller im erweiterten Modus verwenden, müssen einen *VioSavRedrawWait*- und einen *VioModelWait*-Aufruf ausführen und zu jedem Zeitpunkt zum Speichern und Laden der Bildschirminformationen bereit sein. *VioSavRedrawWait* kehrt zurück, wenn das Programm die Bildschirmdarstellung speichern oder laden soll; *VioModelWait* kehrt zurück, wenn die Statusinformation des Bildschirm-Controllers geladen werden soll.

Der zweite Teil der Problemlösung ist folgender: Falls die Anwendung die Speicheroperation in 30 Sekunden nicht vervollständigt hat, betrachtet das System die Operation als erfolglos und schaltet den Bildschirm einfach um. Der desaktivierte Prozeß wartet im Hintergrund, damit der Bildschirm nicht unberechtigterweise verändert wird. Nach der Rückschaltung des Prozesses in den Vordergrund erwartet das System die Behandlung des Fehler durch den Prozeß. In diesem Fall kann die Bildschirmdarstellung einer Anwendung zerstört sein. Der Programmierer sollte einen Befehl zur erneuten Darstellung der Bildschirmanzeige in dieser Situation zur Verfügung stellen. Im denkbar ungünstigsten Fall ist die aktuelle Anwendung zerstört, das System funktioniert aber noch mit allen anderen Prozessen.

Die Entwicklung eines Systems, das Fehler für eine Anwendung entdecken und korrigieren kann, ist nicht möglich. Ein System kann lediglich so entwickelt werden, daß ein nicht funktionierendes Programm möglichst wenig Einfluß auf die restlichen Prozesse nimmt.

Es ist zu hoffen, daß die lange und umfangreiche Erläuterung der Regeln, Probleme und Problemlösungen bei der Verwendung des Presentation Manager und der VIO Sie nicht abgeschreckt hat. Sie brauchen sich über diese Eigenheiten nur dann Gedanken zu machen, wenn Anwendungsprogramme auf den Bildschirm direkt zugreifen und damit die Schnittstellen VIO und Presentation Manager umgehen. Bei der Entwicklung einfacher Textverarbeitungssysteme, die das VIO-Paket oder den Presentation Manager verwenden (gleiches gilt für Graphikanwendungen, die mit dem Presentation Manager arbeiten), können diese Überlegungen ebenfalls entfallen. VIO und Presentation Manager behandeln die Speicherung von Bildschirmdarstellungen und unterstützen die Aktualisierung von Bildschirmdarstellungen auch im Hintergrund. Abschließend ist noch zu sagen, daß sogar Prozesse, die Nummernoperationen (Schreibvorgänge) mit STDOUT ausführen, die VIO- bzw. Presentation Manager-Schnittstelle ebenfalls verwenden. Wenn STDOUT auf die Bildschirmeinheit zeigt, lenkt das Betriebssystem STDOUT auf VIO bzw. Presentation Manager um. Für das Anwendungsprogramm ist dies nicht erkennbar; auch die Vorder- und Hintergrundverarbeitung muß nicht berücksichtigt werden. Gleiches gilt für Bildschirmmodi, Ladevorgänge bei Hardwarefehlern und ähnlichem.

Kapitel 14

Interaktive Programme

Die Mehrzahl der Anwendungsprogramme für PCs stehen über die Schnittstellen Bildschirm und Tastatur mit dem Benutzer in Verbindung. Der Grund: Die meisten Arbeitsplatzcomputer dienen in erster Linie der Ausführung interaktiver Anwendungen. Daher enthält OS/2 eine Vielzahl von Serviceroutinen für die Gestaltung der Schnittstelle zwischen Programm und Benutzer.

In den vorangegangenen Kapiteln wurde die Funktion des Presentation Manager als Kommunikationsschnittstelle für Anwendungsprogramme deutlich. Alternativ steht das VIO-Paket bereit. Weiterhin besteht natürlich die Möglichkeit, daß in einem Programm die Benutzerschnittstelle ohne besondere Unterstützung durch das Betriebssystem implementiert wird. Durch den Einsatz des Presentation Manager werden viele Problemfälle von vornherein umgangen, so etwa die Zuordnung der Tastatureingaben an die verschiedenen Prozesse einer Bildschirmgruppe.

Wenn Sie nur Anwendungsprogramme mit dem Presentation Manager entwickeln wollen, können Sie dieses Kapitel überspringen. Bei interaktiven Anwendungen, die nicht auf den Presentation Manager ausgerichtet sind, muß der Programmierer die Einbindung der Interaktionsmerkmale des Betriebssystems in der Ein-/Ausgabearchitektur verstehen. Nur dadurch läßt sich erreichen, daß Anwendungsprogramme in allen denkbaren Situationen korrekt ausgeführt werden.

14.1 Ein-/Ausgabearchitektur

Die Ein-/Ausgabearchitektur von OS/2 sieht vor, daß alle Programme Eingaben aus der Einheit mit der STDIN-Nummer lesen und die Ausgaben in die Einheit mit der STDOUT-Nummer schreiben. Dies gilt für alle Anwendungen, insbesondere aber für interaktive Programme. Der Grund ist darin zu sehen, daß OS/2 und die Hilfsprogramme zur Programmausführung wie CMD.EXE (*shell*-Programme) für die Zugriffssteuerung auf Bildschirm und Tastatur mit STDIN und STDOUT arbeiten. Falls beispielsweise zwei Prozesse gleichzeitig von der Tastatur lesen, werden einige Tastenanschläge an den einen Prozeß übergeben und die anderen gehen in einer nicht vorhersagbaren Weise an den anderen Prozeß. Auch sollten nicht mehrere Prozesse gleichzeitig auf den Bildschirm schreiben (in derselben Bildschirmgruppe und/oder im gleichen Fenster; Anwendungen, die verschiedene virtuelle Bildschirmgruppen ansprechen, können ihre eigene Bildschirmdarstellung ohne Rücksicht auf andere virtuelle Bildschirme adressieren). In einer Bildschirmgruppe sollen weder zu viele noch zu wenige Prozesse Tastatur- und Bildschirmzugriffe ausführen. CMD.EXE sollte immer eine Eingabeaufforderung ausgeben können, auch wenn ein Benutzer eine interaktive Anwendung in einer Bildschirmgruppe beendet hat (zum Beispiel ein Programm, das von CMD.EXE ausgeführt wurde).

Wie wird diese Situation behandelt? In einer Bildschirmgruppe können viele verschiedene Prozesse gleichzeitig ablaufen. So kann CMD.EXE beispielsweise ein Tabellenkalkulationsprogramm ausführen, das eine Untershell aufruft (eine Kopie von CMD.EXE). Der Benutzer führt gleichzeitig ein Stapelverarbeitungsprogramm aus, das wiederum zum Aufruf eines Editors führt. Der Editor bringt einen C-Compiler zur Ausführung, um C-Quellcode nach Fehlern zu durchsuchen. Hinzu kommt, daß die CMD.EXE-Shell der obersten Ebene eine Assemblerroutine ausführt (mit der Anweisung DETACH ähnlich wie & unter Unix).

Häufig laufen in der Bildschirmgruppe viele verschiedene Prozesse ab, einige sind interaktiv, andere nicht. Dabei spricht jedoch immer nur ein Prozeß die Tastatur und den Bildschirm an. Es würde sich anbieten, immer dem zuletzt ausgeführten Prozeß den Tastatur- und Bildschirmzugriff zuzuordnen. Das ist aber nicht sinnvoll: Das zuletzt ausgeführte Programm ist der C-Compiler, mit diesem kann man aber nicht interaktiv arbeiten. OS/2 kann nicht entscheiden, welcher Prozeß den Bildschirm und die Tastatur ansprechen darf, da das Betriebssystem keine Kenntnis über die Funktionen der einzelnen Prozesse besitzt. OS/2 kennt lediglich die Eltern-Kind-Beziehungen. Diese Informationen reichen jedoch in komplexen Situationen nicht für die sinnvolle Zuordnung des Bildschirm- und Tastaturzugriffs aus.

Da OS/2 den Zugriff auf Bildschirm und Tastatur nicht verwalten kann, müssen die Prozesse selbst Festlegungen treffen. Die Regel ist einfach: Der Prozeß, der gerade den Zugriff auf Bildschirm und Tastatur besitzt, kann diesen an einen Kindprozeß abtreten oder für sich selbst behalten. Beim Abtreten der Zugriffserlaubnis an einen Kindprozeß muß der Vaterprozeß mit der Benutzung des Bildschirms und der Tastatur bis zur Beendigung des Kindprozesses warten. Nach der Übergabe der Zugriffserlaubnis an den Kindprozeß hat der Vaterprozeß keinen Einfluß mehr auf die weitere Zugriffsvergabe. Der Kindprozeß kann die Zugriffsberechtigung auf einen anderen Kindprozeß übertragen oder sie selbst nutzen. Der Vaterprozeß sollte den Zugriff auf Bildschirm und Tastatur meiden, bis der Kindprozeß und dessen Ablegerprozesse beendet sind.

Im folgenden finden Sie ein Beispiel dafür. Falls CMD.EXE der erste Prozeß einer Bildschirmgruppe ist, wird STDIN auf die Tastatur gerichtet und mit STDOUT wird die Einheit *Bildschirm* geöffnet. CMD.EXE startet die Ausführung eines Tabellenkalkulationsprogramms, ohne daß es erkennen kann, ob das Tabellenkalkulationsprogramm interaktiv ist. Es übergibt dem Kindprozeß (hier dem Tabellenkalkulationsprogramm) die Nummern STDIN und STDOUT, die auf Tastatur und Bildschirm gerichtet sind. Da CMD.EXE den Zugriff auf Bild-

schirm und Tastatur an den Kindprozeß übergeben hat, kann bis zur Beendigung des Kindprozesses von CMD.EXE weder STDIN noch STDOUT verwendet werden. An diesem Punkt gibt CMD.EXE üblicherweise einen *DosCWait*-Aufruf aus, um die Beendigung des Kindprozesses zu erwarten.

Nach dem Aufruf des Tabellenkalkulationsprogramms schreibt dieses nach STDOUT (Bildschirm) und liest aus STDIN (Tastatur). Die Anweisung zur Ausführung von CMD.EXE durch das Tabellenkalkulationsprogramm hat zur Folge, daß die neue Kopie von CMD.EXE als interaktiv angesehen wird und die STDIN- und STDOUT-Nummern erbt. Das Tabellenkalkulationsprogramm muß die Ein- und Ausgabe mit STDIN und STDOUT vermeiden bis der Kindprozeß - CMD.EXE - beendet ist. Solange Prozesse interaktive Kindprozesse aufrufen, bleibt dieses Schema erhalten. Nach dem Abbruch der ersten Kindprozesse werden Bildschirm- und Tastaturzugriff in umgekehrter Reihenfolge durch die Prozeßreihe zurückgegeben.

Was aber ist mit dem über den Befehl DETACH ablaufenden Assembler, den CMD.EXE vor der Ausführung der Tabellenkalkulation gestartet hat? In diesem Fall hat der Benutzer CMD.EXE explizit mitgeteilt, daß die Anwendung von der Tastatur *abgehängt* (*detached*) ausgeführt werden soll. Hat der Benutzer die Nummer STDIN für den Assembler spezifiziert (zum Beispiel eine Datei), verwendet CMD.EXE diese für Kindprozesse, die die STDIN-Nummer erben. Wurde vom Benutzer keine Alternativeinheit für STDIN angegeben, öffnet CMD.EXE STDIN mit der NULL-Einheit, so daß die Anwendung bei einem Lesevorgang das EOF-Zeichen empfängt. In diesem Fall kann ein Kindprozeß STDIN nicht erben (CMD.EXE wurde mitgeteilt, daß die Anwendung die Tastatur nicht ansprechen wird), CMD.EXE kann also weiterhin den Zugriff auf Tastatur und Bildschirm behalten. Es wird erneut eine Eingabeaufforderung auf dem Bildschirm dargestellt und auf die Eingabe neuer Befehle gewartet. In Bild 14.1 wird ein typischer Prozeßbaum gezeigt. Die schattiert dargestellten Prozesse haben die STDIN-Nummer geerbt, die auf die Tastatur zeigt, gleiches gilt für die STDOUT-Nummer, die den Bildschirm adressiert. Alle Prozes-

se, die Tastatur-/Bildschirmzugriffsberechtigungen weitergegeben haben, müssen in einem einzigen Pfad liegen, wenn die zuvor besprochenen Regeln befolgt wurden, da jeder Prozeß den Zugriff auf Tastatur und Bildschirm nur dann weitergeben kann, wenn er die Zugriffsberechtigung von seinem Vaterprozeß erhalten hat. Die Zugriffsberechtigung kann stets nur an jeweils einen Kindprozeß weitergegeben werden.

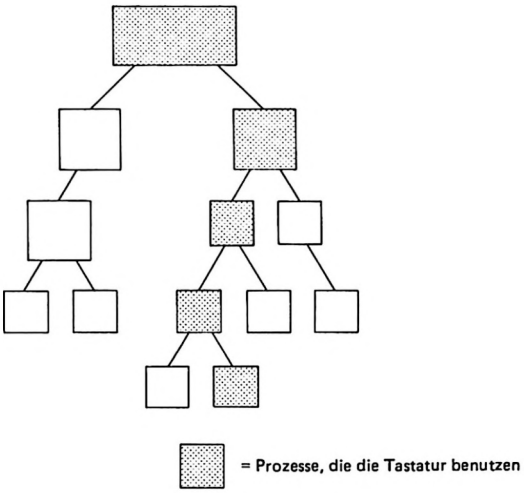


Bild 14.1 Prozesse, die die Tastatur verwenden

Ein Problem tritt bei der Verwendung hochauflösender Graphikanwendungen auf. Bis zu diesem Punkt wurde nur das Modell der Architektur erläutert, dem man folgen muß, um Probleme im Zusammenhang mit Bildschirm und Tastatur systemweit zu vermeiden. Anwendungsprogramme können und sollen besondere Services zur Optimierung der interaktiven Ein-/Ausgabe verwenden, solange dies nicht dem zugrundeliegenden Design widerspricht. OS/2 stellt dazu die KBD-, VIO- und Presentation Manager-Dynlink-Pakete bereit. Die Programme mit

optimierter Ablaufgeschwindigkeit stellen eine direkte Schnittstelle zur Hardware dar und umgehen die nur beschränkt einsetzbare Schnittstelle STDIN/STDOUT. Der Schlüssel hierbei ist der direkte Zugriff auf die Hardware: Ein Prozeß soll hardwarespezifische Schnittstellen zur Optimierung der Ablaufgeschwindigkeit einsetzen, solange sichergestellt ist, daß das Architekturmodell den Zugriff auf die Einheit erlaubt.

In der Praxis ist dies einfach. Jedes interaktive Programm, das die KBD-Schnittstelle ansprechen will, muß Sorge dafür tragen (über *DosQHandType*), daß seine STDIN-Nummer mit der Tastatur verbunden ist. Zeigt STDIN nicht auf die Tastatur, ist diese einem anderen Programm zugeordnet und das interaktive Programm darf die KBD-Schnittstelle zu diesem Zeitpunkt nicht einsetzen. Die Dynlink-Einheitenschnittstellen prüfen nicht, ob Zugriffe berechtigt sind. Die Anwendung muß für die Gültigkeit ihrer Anforderung selbst Sorge tragen. Dasselbe gilt für STDOUT, die Tastatureinheit und das VIO-Paket. Anwendungen müssen dafür sorgen, daß STDIN und STDOUT auf Tastatur und Bildschirm zeigen, bevor sie eine direkte Schnittstelle ansprechen (wie den Presentation Manager, VIO, KBD, MOU oder den direkten Zugriff auf die Einheit).

Was kann ein interaktives Programm tun, das herausgefunden hat, daß STDIN nicht auf die Tastatur und/oder STDOUT nicht auf den Bildschirm zeigt? Dieses Problem muß vom Programmierer der Anwendung gelöst werden. Einige nicht-interaktive Anwendungen werden dadurch in ihrer Ausführung nicht gestört. So kann beispielsweise der Makroassembler von Microsoft nach den Namen der Quell-, Objekt- und List-Dateien fragen. Obwohl der MASM technisch gesehen mit dem Benutzer kommuniziert, stellt der Assembler keine eigentlich interaktive Anwendung dar (er ist nicht auf die Interaktion mit dem Benutzer angewiesen). Wenn STDIN auf eine Datei zeigt, kann der Assembler die Antworten auch aus der Datei lesen. Er braucht dabei nicht zu prüfen, ob STDIN auf die Tastatur zeigt, da er das KBD-Paket nicht verwendet. Statt dessen liest MASM die Namen der zu bearbeitenden Dateien aus STDIN und versucht die Ausführung mit den von dort kommenden Daten.

Andere Programme benötigen vielleicht keine interaktive Schnittstelle; wenn sie aber eine Interaktion durchführen, so sollen die KBD- und VIO-Pakete zur Beschleunigung des Datendurchsatzes verwendet werden. Diese Anwendungen müssen prüfen, ob STDIN und STDOUT auf die entsprechenden Einheiten gerichet sind. Ist dies der Fall, können die Anwendungen die Beschränkungen von STDIN und STDOUT umgehen und die Pakete KBD bzw. VIO verwenden. Falls die Nummern nicht auf die richtigen Einheiten zeigen, sollte eine Fehlermeldung an STDERR ausgegeben und das Programm beendet werden. Gleiches gilt für Anwendungen, die auf jeden Fall einen Dialog führen müssen. Falls der Benutzer eine interaktive Anwendung wie einen WYSIWYG-Editor mit dem Befehl *detach* ausführen läßt, ist die Ausgabe einer Fehlermeldung vor dem Löschen der Bildschirmdarstellung anderer Prozesse vorzuziehen. Die Bildschirmgruppe wird nicht verwendbar und der Benutzer ist unter Umständen nicht in der Lage, den Editor zu beenden, der den Fehler hervorgerufen hat, da die Tastaturzuweisung nicht mehr eindeutig nachvollzogen werden kann.

Es ist technisch möglich, den Zugriff auf die Tastatur zu beschränken, dies wird aber im allgemeinen vermieden. Üblicher ist, daß Anwendungen Zugriff auf den Bildschirm, nicht aber auf die Tastatur besitzen. Obwohl die meisten Benutzer Fehlermeldungen auf dem Bildschirm (beispielsweise von Compilern) als störend empfinden, kann dies im Einzelfall erwünscht sein. Die unmittelbare Mitteilung eines Fehlers kann zwar mit einer zerstörten Bildschirmdarstellung enden, was dem Benutzer aber unter Umständen nicht so wichtig ist. Für die Verwendung der Pakete VIO, KBD und dem Environment Manager müssen die Zuordnungen von STDIN und STDOUT überprüft werden.

Es wurde bereits beschrieben, wie Anwendungen Kindprozesse initiieren, die den Zugriff auf Bildschirm und Tastatur erben. Dies geschieht wie folgt: Falls ein Vaterprozeß keinen Zugriff auf die Tastatur hat, kann er bei der Ausführung eines Kindprozesses (mit *DosExecPgm*) die Zugriffsberechtigung auch nicht vererben. Für einen *DosExecPgm*-Aufruf muß der Vater-

prozeß den Status von STDIN nicht überprüfen. Es gelten die folgenden Regeln:

Ausführung von Programmen

- Erbt der Kindprozeß die STDIN-Nummer, darf der Vaterprozeß bis zur Beendigung des Kindprozesses keinen Zugriff über diese Nummer ausüben.
- Falls der Kindprozeß die STDIN-Nummer nicht erben soll (das aufrufende Programm gibt die Zugriffsberechtigung nicht ab), muß die STDIN-Nummer des Kindprozesses auf eine Datei oder die NULL-Einheit gerichtet werden. Verlassen Sie sich nicht darauf, daß der Kindprozeß die Nummer nicht verwendet; der Kindprozeß könnte einen anderen Kindprozeß aufrufen (mit *DosExecPgm*), der wiederum auf Bildschirm und Tastatur zugreift.
- Ein Prozeß kann nur einem Kindprozeß die STDIN-Nummer vererben. Falls ein Prozeß mehrere Kindprozesse parallel ausführen läßt, kann STDIN nur an einen dieser Folgeprozesse weitergegeben werden. Die anderen Kindprozesse müssen andere STDIN-Quellen verwenden.
- Diese Regeln beziehen sich auf die STDIN-Nummern, die auf Pipes oder Dateien sowie auf die Tastatur gerichtet sind. Das Anwendungsprogramm muß die Herkunft von STDIN nicht überprüfen.

Alle Prozesse

- Vor der Verwendung des KBD-Paketes muß überprüft werden, ob die STDIN-Nummer auf die Tastatur zeigt; Gleiches gilt für den *Session Manager* oder SIGCTLC (siehe unten). Die direkten Zugriffsleistungsmerkmale auf Einheiten sollten nicht verwendet werden, wenn STDIN nicht auf die Tastatur gerichtet ist.
- Vor der Verwendung des VIO-Paketes und des Session Manager muß sichergestellt werden, daß STDOUT auf den Bildschirm zeigt. Diese Zugriffsleistungsmerkmale auf Ein-

heiten sollten nur zur Anwendung kommen, wenn STDOUT auf den Bildschirm gerichtet ist.

- Falls ein Prozeß einen Kindprozeß ausführt, der STDIN erbt, darf der Vaterprozeß nicht vor dem Kindprozeß beendet werden. Der Vaterprozeß nimmt nämlich an, daß die Beendigung des direkt angebundenen Kindprozesses der Wiederverfügbarkeit der STDIN-Einheit gleichkommt.

14.2 Ctrl-C und Ctrl-Break

Ein weiterer Aspekt verdient besondere Beachtung: Die Bearbeitung der Tastenkombinationen Ctrl-C und Ctrl-Break durch das Betriebssystem. Die beiden Ereignisse beziehen sich auf die Tastaturhardware. Das grundlegende Problem ist hier schnell aufgezeigt: Mit der Betätigung der Tastenkombinationen Ctrl-C oder Ctrl-Break durch den Benutzer ist das Betriebssystem gezwungen, zu reagieren. Es liegt auf der Hand, daß ein Prozeß oder mehrere Prozesse oder sogar ein gesamter Unterbaum von Prozessen beendet oder ein Signal an diese Elemente übergeben werden muß. Was ist zu unternehmen (Ausgabe eines Signals oder Prozeßbeendigung)? Und welche Prozesse sollen angesprochen werden?

OS/2 definiert eine Konvention, die den Prozessen selbst die Entscheidung überläßt. Stellen Sie sich eine Anwendung zur Ausführung von Befehlen (eine *Befehlsanwendung*) vor, die im Befehlsmodus abläuft. Im Befehlsmodus liest eine Befehlsanwendung einen Befehl, führt diesen aus und kehrt üblicherweise in den Befehlsmodus zurück. Es kann also vorkommen, daß die Betätigung der Tastenkombination Ctrl-Break die Befehlsanwendung zur Beendigung des derzeitigen Befehls und zur Rückkehr in den Befehlsmodus veranlassen soll. Dies wird bei den meisten interaktiven Programmen geschehen, für Programmanwendungen, die nicht interaktiv ablaufen, ist dies aber ungewöhnlich. Durch die Eingabe von MASM hinter der Eingabeaufforderung von CMD.EXE wird das Programm MASM von CMD.EXE als Kindprozeß ausgeführt. CMD.EXE ist eine Befehlsanwendung, MASM aber nicht. Diese Unterscheidung wird

nicht von OS/2 getroffen, sondern soll hier nur der Erläuterung dienen.

Gemäß den Systemkonventionen bedeuten die Tastenkombinationen Ctrl-C und Ctrl-Break *Unterbrich, was gerade getan wird*. OS/2 erzeugt Signale als Antwort auf die Tastenkombinationen Ctrl-C und Ctrl-Break, es wird kein Prozeß unmittelbar beendet. OS/2 erkennt, an welchen Prozeß das Signal übergeben werden muß: Dies ist der letzte Befehlsprozeß im Prozeßbaum der korrespondierenden Bildschirmgruppe. Dies ist nicht einfach - wie soll OS/2 feststellen, welcher Prozeß anzusprechen ist? Ein vollständiger Prozeßbaum in einer Bildschirmgruppe kann ein komplexes Erscheinungsbild besitzen; einige Prozesse sind bereits beendet, wodurch mehrere Unterbäume entstanden sind, die aufgrund der Unterbrechungen (beendete Kindprozesse, die noch laufende Kindprozesse haben) voneinander unabhängig sind.

Während sich der vollständige Prozeßbaum recht komplex gestalten kann, ist der Prozeßunterbaum, dem der Tastaturzugriff zugeordnet ist, aber wesentlich einfacher (da immer nur ein Kindprozeß den Zugriff erben kann). Da ein Prozeß den Tastaturzugriff nur erben und die Tastatur nicht explizit öffnen kann, gibt es im System nur einen Pfad, der Tastaturzugriff besitzt. Unterbrechungen in diesem Pfad kann es nicht geben, da ein Vaterprozeß nicht vor der Beendigung des Kindprozesses, der den Tastaturzugriff erhalten hat, beendet werden darf. Aus diesem Grund braucht OS/2 lediglich einen Befehlsprozeß im Befehlsunterbaum zu finden und dessen nachfolgende Prozesse nach weiteren Kindprozessen zu untersuchen. Der letzte Prozeß im Ast erhält das Signal. Genau genommen verwendet OS/2 einen effizienteren Algorithmus zur Erledigung dieser Aufgabe, die gegebene Erklärung soll lediglich aufzeigen, wie einfach die Suche nach dem richtigen Befehlsprozeß ist.

Hinweis: Eigentlich sollte ein Prozeß den Tastaturzugriff nur erben. Die Tastatureinheit kann explizit geöffnet werden. Dieser Vorgang führt zu einem schwerwiegenden Fehler, wenn die ererbte STDIN-Nummer nicht bereits auf die Tastatureinheit zeigt.

Bild 14.2 zeigt mögliche Prozeßbäume. Die schattiert dargestellten Prozesse haben die Nummern auf Tastatur und Bildschirm ererbt. Bei den mit C markierten Prozessen handelt es sich um Befehlsprozesse.

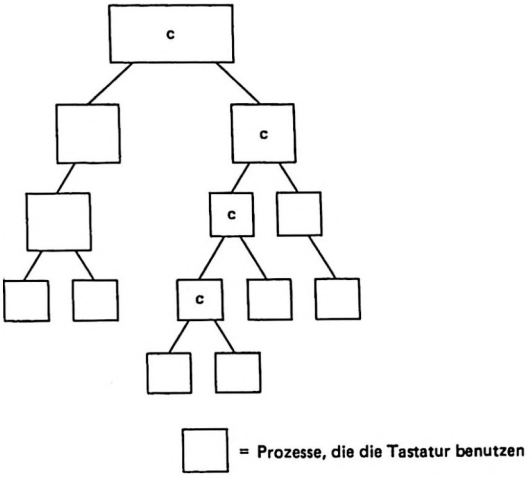


Bild 14.2 Ctrl-C-Route in einem Prozeßbaum

Dies führt zu einer letzten Frage: Wie erkennt OS/2, ob eine Anwendung ein Befehlsprozeß ist? Befehlsprozesse bzw. Befehlsanwendungen haben eine andere Aufgabe als andere Prozesse. Da definitionsgemäß ein Befehlsprozeß durch die Betätigung der Tastenkombination Ctrl-C oder Ctrl-Break nicht abgebrochen werden soll, verfügen alle Befehlsprozesse über signalbearbeitende Routinen für diese Tastenkombination. Da die Tastenkombinationen Ctrl-C und Ctrl-Break von Befehlsprozessen immer abgefragt werden, muß definiert werden, daß *nur* Befehlsprozesse dies tun. Das knüpft an die Ausführungen über die Prüfung von STDIN vor der Verwendung der Tastatur an. Die Betätigung der Tastenkombinationen Ctrl-C und Ctrl-Break ist eine Form der Ein-/Ausgabe über die Tastatur. Die Umlei-

tung der Tastenkombination an eine signalbearbeitende Routine darf nur erfolgen, wenn zuvor durch das Programm sichergestellt wurde, daß STDIN auf die Tastatur (da dies eine Form der Ein-/Ausgabe mit der Tastatur ist) gerichtet ist. Die Abfrage der Tastenkombinationen sollte so gelegt werden, daß dem Programm der Abbruch oder eine entsprechende Aktion problemlos möglich ist. Das Signal SIGTERM oder besser der *Exit-List*-Mechanismus stellt dieses Leistungsmerkmal bereit und verbirgt Gründe zur Beendigung von Prozessen, die nicht auf die Tastatur zurückzuführen sind. Auf diese Weise haben alle Prozesse mit Ctrl-C- und Ctrl-Break-Abfrage Zugriff auf die Tastatur und werden beendet, sobald eine der Tastenkombinationen auftritt.

Bislang wurde ausführlich erläutert, wie OS/2 den Prozeß findet, dem ein Ctrl-C-Signal zugeordnet werden soll, und wie der Prozeß bei Auftreten des Signals zu reagieren hat. Die Systemkonventionen hierfür sollten befolgt und die aktuell bearbeitete Aufgabe möglichst schnell unterbrochen werden. Falls die Anwendung keinen Befehl abarbeitet, sollte der Tastaturpuffer gelöscht und eine erneute Eingabeaufforderung angezeigt werden. Beim Auftreten der Tastenkombination, die das Signal auslöst, und der gleichzeitigen Abarbeitung eines Befehls, der in der Anwendung implementiert ist, springt die Befehlsausführung von der signalbearbeitenden Routine zur Befehlseingabeschleife oder setzt eine Flagge zur vorzeitigen Beendigung des gerade ausgeführten Befehls. Es kann vorkommen, daß die vorzeitige Beendigung eines Befehls Daten vernichtet. In diesem Fall darf die Befehlsausführung nicht unterbrochen werden.

Führt eine Anwendung einen Kindprozeß aus, wird dieser üblicherweise durch einen *DosKill*-Aufruf (der bezüglich des Kindbefehlsunterbaums ausgeführt wird) abgebrochen. Daher kann mit der Tastenkombination Ctrl-C ein Programm - siehe Makroassembler - beendet werden. Ctrl-C wird an den aufrufenden Prozeß von MASM gesendet, der ein Befehlsprozeß ist und ein *DosKill*-Aufruf an MASM ausgibt (häufig ist CMD.EXE der aufrufende Prozeß, andere Programme können aber zwischengeschaltet sein). Das Programm MASM führt (sofern er-

forderlich) einen *ExitList*-Aufräumvorgang aus (beispielsweise zum Löschen von Zwischendateien) und wird beendet. Der Befehlsprozeß, der den Makroassembler ausgeführt hat (üblicherweise CMD.EXE), erkennt die Beendigung und gibt die Zeichenfolge *^C* gefolgt von einer neuen Eingabeaufforderung auf dem Bildschirm aus.

Kapitel 15

Das Dateiverwaltungssystem

Das Dateisystem von OS/2-Version 1.0 unterscheidet sich nur geringfügig von dem des Betriebssystems MS-DOS. Dies erklärt sich in erster Linie aus der gewünschten Kompatibilität zu MS-DOS-Programmen. Auf der anderen Seite waren die Zeitvorgaben zur Entwicklung von OS/2 recht knapp. Der Zeitrahmen war nicht ausreichend, um in der ersten OS/2-Version alle gewünschten Leistungsmerkmale unterzubringen, Erweiterungen des Dateisystems wurden daher für künftige Versionen geplant. Im folgenden will ich eine Erklärung liefern, weshalb ein so wichtiger Punkt wie ein neues Dateiverwaltungssystem aufgeschoben wurde.

Die Mikrocomputerbranche entwickelte sich in zwei Richtungen: *Massensoftware* und *Standards*. Die "massenhafte" Vermarktung von Software erlaubt den Kauf hochentwickelter Programme für relativ wenig Geld - der Ausdruck *wenig Geld* muß hier im Vergleich zu den Entwicklungskosten gesehen werden, die oftmals Millionen Dollar betragen. Die Massenvermarktung unterstützt die Standardisierung, da kein Benutzer System, Peripherieeinheiten und Geräte kaufen will, die mit den weithin angebotenen Programmen nicht kompatibel sind. Andererseits unterstützt die Akzeptanz der Standards die Entwicklung von Massensoftware, so daß das Absatzpotential für Programme immens zunimmt. In diesem Kreislauf eröffnet sich die Chance zur Entwicklung äußerst komplexer Anwendungen, da die hohen Gestehungskosten durch einen breiten Massenmarkt abgefangen werden.

Der Synergieeffekt zwischen Standardisierung und Massenvermarktung von Software betrifft natürlich auch die Betriebssystementwicklung. Auf den ersten Blick scheint die Erweiterung eines Betriebssystems problemlos. Die Entwickler bauen neue Leistungsmerkmale in eine neue Version ein und die Anwendungen werden zur Nutzung der neuen Leistungsmerkmale umgeschrieben. Mit der Massenvermarktung von Software ist dies allerdings nicht möglich. Microsoft könnte eine neue Version von OS/2 mit einigen neuen Leistungsmerkmalen herausbringen (und tatsächlich wird dies auch der Fall sein), anfänglich würden aber nur wenige Anwendungen die neuen Leistungsmerkmale nutzen. Dies beruht auf der begrenzten Marktdurchdringung der neuen Version. Nach einer bestimmten Zeit nach der Verfügbarkeit der neuen Version haben vielleicht zehn Prozent der OS/2-Benutzer ihre Version aktualisiert. Ein ISV (*Independent Software Vendor*), der ein neues Produkt plant, hofft natürlich auf einen Bestseller. Er muß entscheiden, ob das Produkt die neuen Leistungsmerkmale ansprechen und damit nur zehn Prozent des potentiellen Marktes adressieren soll. Die Alternative besteht in der Verwendung der Leistungsmerkmale der verbreiteten Betriebssystemversion, was das Programm auf allen derzeitigen Systemen ausführbar macht (inklusive der zehn Prozent, die neueste OS/2-Versionen verwenden). Im allgemeinen werden Softwareanbieter die wenig verbreiteten Leistungsmerkmale eines Betriebssytems in ihrem Programm nicht einsetzen, bis die Mehrheit der existierenden Systeme diese unterstützen.

Der Schlüssel zur Einführung neuer Leistungsmerkmale in einem Betriebssystem ist also nicht dessen Verfügbarkeit oder Nützlichkeit. Der Schlüssel besteht darin, daß die Version mit den neuen Leistungsmerkmalen möglichst rasch eine große Marktdurchdringung erlangt. Falls dies nicht der Fall ist, wird das neue Leistungsmerkmal nicht verwendet. Aus diesem Grunde wurden neue MS-DOS-Versionen nur herausgegeben, wenn die Unterstützung neuer Hardwareelemente erforderlich wurde. MS-DOS Version 2.0 wurde für die Produktlinie des IBM XT benötigt; MS-DOS Version 3.0 muß der AT-Produktreihe zugerechnet werden. OS/2 stellt keine Ausnahme dar: Das Betriebs-

system soll den *Protect*-Modus der Rechner mit dem Intel-Prozessor 80286 unterstützen. Falls eine neue Version eines Systems nur neue Leistungsmerkmale enthält, die nicht unbedingt gebraucht werden, ist die Marktdurchdringung anfänglich sehr niedrig und verbessert sich nur langsam. Die Kosten zur Aktualisierung sind nur gerechtfertigt, wenn die neuen Leistungsmerkmale unabkömmlich sind. Anwendungsprogramme fordern die Verfügbarkeit neuer Leistungsmerkmale aber häufig erst, wenn die Verbreitung groß genug ist.

In den Augen der Entwickler ist die erste Version von OS/2 damit sehr wichtig. Durch sie ist eine Möglichkeit zur Einführung neuer Leistungsmerkmale in den Betriebssystemstandard der PCs gegeben. Dieses "Fenster" wird für lange Zeit nicht mehr so weit geöffnet sein.

Aus diesem Grund wurde die Erneuerung des Dateiverwaltungssystems zurückgestellt: Es kann in späteren Versionen erweitert und verbessert werden, wobei existierende Anwendungen ohne Änderungen weiterhin unterstützt werden. Viele andere OS/2-Leistungsmerkmale mußten dagegen unbedingt in die erste Version des Betriebssystems aufgenommen werden, da sie sonst für weit verbreitete Anwendungen niemals verfügbar werden würden.

15.1 Das OS/2-Dateiverwaltungssystem

Einige kleinere Änderungen wurden am Dateiverwaltungssystem von OS/2 Version 1.0 durchgeführt. Zwei davon sind der Erwähnung wert: Die erste ist die asynchrone Ein- und Ausgabe, die zwei Funktionen umfaßt (*DosReadAsync* und *DosWrite-Async*). Die Funktionen entsprechen den *DosRead*- und *DosWrite*-Aufrufen mit der Ausnahme, daß sie die Kontrolle sofort an das aufrufende Programm zurückgeben (üblicherweise vor der Vervollständigung der Ein-/Ausgabeoperation). Die Funktionen übernehmen die Nummer einer Semaphore, die nach der Beendigung der Ein-/Ausgabeoperation gelöscht wird. Die Threads des aufrufenden Prozesses können die Semaphore zur Überprüfung der Ein-/Ausgabebeendigung verwenden. Die Funktion

DosMuxSemWait ist in diesem Zusammenhang besonders nützlich, da sie das Warten auf mehrere Semaphorenereignisse erlaubt, die auch asynchrone Ein-/Ausgabe-, IPC- und Zeitgebereereignisse beinhalten dürfen.

Das zweite neue Leistungsmerkmal des Dateiverwaltungssystems ist eine erweiterte Partitionierung: Die Unterstützung umfangreicher physikalischer Laufwerke in Form von mehreren Bereichen ist gewährleistet, wobei mehrere Partitionen FAT-Dateiverwaltungssysteme enthalten können. Die Folge ist, daß OS/2 eine große Festplatte als zwei oder mehrere kleinere Einheiten betrachten kann, von denen jede für sich der Größenbeschränkung des Dateiverwaltungssystems unterliegt. Der Aberglaube, daß MS-DOS nur Laufwerke bis zu einer Größe von 32 Mbyte verwenden kann, ist weit verbreitet (allerdings nicht richtig). Die Begrenzung liegt darin, daß ein Laufwerk in nicht mehr als 65.535 Sektoren aufgeteilt werden kann. Aus der Standardsektorgröße von 512 Bytes ergibt sich dann der Wert von 32 Mbyte. Hinzu kommt, daß jeder Datenträger auf 32.768 Cluster begrenzt ist. Ein *Sektor* ist eine Einheit des Diskettenspeicherplatzes; Disketten können immer nur in ganzen Sektoren gelesen und geschrieben werden. Die Größe der Sektoren wird bei der Formatierung des Datenträgers festgelegt. Ein *Cluster* ist die Einheit, in der die Speicherkapazitätsbelegung für Dateien und Verzeichnisse erfolgt und kann aus einem oder mehreren vollständigen Sektoren bestehen. Das MS-DOS-Dateiverwaltungssystem erlaubt ein Maximum von 65 Kbyte pro Sektor, während nur 32 Kbyte für Cluster erlaubt sind. Folglich muß eine 32-Mbyte-Platte in Clustern zu zwei (oder mehr) Sektoren belegt werden. Die Entwicklung von Einheitentreibern, die als Sektorgröße ein Mehrfaches von 512 Bytes unterstützen, erlaubt die Umgehung der 65-Kbyte-Sektorbeschränkung und dadurch die Verwendung einer Festplatte, die mehr als 32 Mbyte Kapazität enthält. Der für MS-DOS beschriebene Trick kann auch unter OS/2 angewendet werden, besitzt aber den Nachteil, daß dies eine Vergrößerung der Sektoren mit sich bringt, wodurch viel Speicherkapazität verschwendet wird.

Hinweis: Das FAT-Dateiverwaltungssystem kann mit einem Maximum von 32-Kbyte-Belegungseinheiten (Clustern) arbeiten. Unabhängig von der Größe des Datenträgers müssen alle Dateien eine Mindestdiskettenkapazität von 1/32 Kbyte des Gesamtdatenträgers belegen. Dies bedeutet, daß eine 60-Mbyte-Festplatte unter Verwendung von 1024-byte-Sektoren Einheiten mit einer Größe von 2048 Bytes belegt.

Die erweiterte Leistungsfähigkeit zur Partitionierung eines Datenträgers unter OS/2 Version 1.0 stellt eine Zwischenlösung dar, die den Kapazitätsverlust durch die interne Fragmentierung reduziert: Mehr als eine Partition kann ein FAT-Dateiverwaltungssystem enthalten. Partitionierte Datenträger unter MS-DOS dürfen nur eine Partition enthalten, die ein MS-DOS-Dateiverwaltungssystem (das heißt, eine FAT) besitzt. Diese Einschränkung wurde unter OS/2 aufgehoben, so daß beispielsweise eine 60-Mbyte-Festplatte in zwei separate logische Laufwerke partitioniert werden kann (beispielsweise C und D), von denen jede eine Speicherkapazität von 30 Mbyte umfaßt.

15.2 Verwaltung der Datenträgernamen

Die Multitasking-Fähigkeit von OS/2 macht eine Erweiterung des Dateiverwaltungssystems im Bereich der Datenträgerkennsätze nötig. Ein Datenträgerkennsatz ist der Name, der der Diskette und den darauf enthaltenen Dateien zugeordnet wird. Ein nichtauswechselbares Laufwerk (Festplatte) enthält stets denselben Datenträger, bei einem Laufwerk mit austauschbaren Datenträgern (wie Diskettenlaufwerken) kann nicht vorhergesagt werden, welche Diskette vom Benutzer zu welchem Zeitpunkt eingelegt wird.

Die Möglichkeit eines Diskettenwechsels wird in Multitasking-Umgebungen zu einem Problem. Verwendet ein Benutzer beispielsweise eine Textverarbeitung und editiert eine Datei auf der Diskette in Laufwerk A, hat der Editor die Datei geöffnet und behält diesen Status bei der Editierung bei. Ohne das Schließen der Datei oder die Beendigung des Editors kann der Benutzer die Bildschirmgruppe umschalten, damit beispielsweise ein Tabellenkalkulationsprogramm abläuft. In diesem Fall muß in Laufwerk A eine andere Diskette eingelegt werden, die Daten zur Bearbeitung der Tabellenkalkulation enthält. Durch die

Umschaltung zum Textverarbeitungsprogramm ohne entsprechenden Diskettenwechsel kommt es für das System zu nicht lösbaren Problemen. Die Betätigung der Taste PgDn läßt das Textverarbeitungssystem einen weiteren Sektor der bereits geöffneten Datei lesen. Das Betriebssystem erkennt - aufgrund der FAT-Informationen in den RAM-Puffern - daß der nächste Sektor der Textdatei der Sektor *N* ist und wird versuchen, den Sektor *N* der falschen Diskette zu lesen (der Diskette mit den Daten für die Tabellenkalkulation). Der gelesene Sektor wird an das Textverarbeitungsprogramm als nächster Sektor der Textdatei zurückgegeben. Dies ist kein schwerwiegender Fehler; noch schlimmer wird es aber, wenn das Textverarbeitungssystem einen Sektor auf die Diskette schreibt, wodurch zwei Dateien vernichtet werden: Die Datendatei der Tabellenkalkulation wird durch den nicht korrekt gesteuerten Schreibvorgang zerstört und Gleiches gilt für die Textdatei, bei der beim nächsten Lesezugriff festzustellen ist, daß die auf die falsche Diskette geschriebenen Daten fehlen.

Diese Probleme können nicht dadurch gelöst werden, daß der Benutzer Disketten mit Bedacht einlegt; viele Programme lesen und schreiben ohne direkten Benutzerbefehl auf/von Disketten. So könnte beispielsweise die Textverarbeitung den aktuellen Text automatisch nach jeweils zwei Minuten speichern. Nach dem Verstreichen dieses Zeitintervalls erfolgt ein vom Benutzer nicht ausdrücklich gewollter Schreibvorgang, auch wenn der Benutzer noch mit der Tabellenkalkulation und der zugehörigen Diskette arbeitet. OS/2 löst diese Probleme, indem Anwendungen ihre Ein-/Ausgaben an geöffnete Dateien nicht an Laufwerk A richten, sondern an einen bestimmten Datenträgerkennsatz (nämlich den, der der Diskette zugeordnet ist, auf der die geöffnete Datei vorliegt). Jeder Datenträgerkennsatz besteht aus einem Datenträgernamen, der im Stammverzeichnis gespeichert ist und einer einzigartigen 32-bit-Datenträgerkennummer, die sich im Startsektor befindet. In Bild 15.1 sind die beiden Teile des Datenträgerkennsatzes gezeigt - einen zur Verwendung durch den Computer (Datenträgerkennummer) und einen für den Bediener (Datenträgername). Jede Dateinummer wird mit

einer bestimmten 32-bit-Datenträgerkennummer verknüpft. Bei einer Ein-/Ausgabeanforderung auf eine Dateinummer überprüft OS/2, ob sich der entsprechende Datenträger im Laufwerk befindet. Dies erfolgt durch den Vergleich des 32-bit-Wertes der Anforderung mit dem des momentan im Laufwerk befindlichen Datenträgers. Bei übereinstimmenden Nummern wird die Operation vervollständigt. Unterscheiden sich die beiden Nummern, verwendet OS/2 den Hardwarefehler-Daemon-Mechanismus, der eine Aufforderung zum Einlegen des richtigen Datenträgers anzeigt.

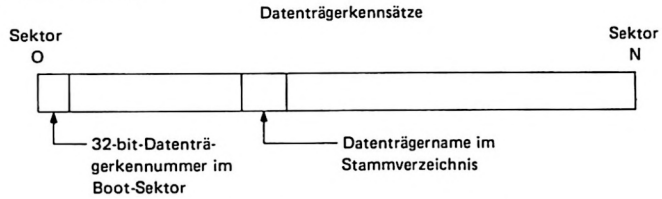


Bild 15.1 Datenträgerkennummer und Datenträgername

Um dies praktikabel zu machen, müssen drei Probleme gelöst werden. Zuerst einmal darf die Überprüfung der Datenträgerkennummer des Datenträgers den Durchsatz nicht mindern. Der Startsektor kann nicht bei jeder Ein-/Ausgabeoperation gelesen werden, wenn dadurch die Ein-/Ausgabegeschwindigkeit um die Hälfte gesenkt wird. Zum zweiten muß sichergestellt sein, daß die Datenträgerkennummer tatsächlich einzigartig ist und drittens muß auch ein Datenträger bearbeitet werden können, der keine Datenträgerkennummern besitzt.

Die Überprüfung des korrekten Datenträgers ist einfach, wenn bekannt ist, wann ein Datenträger gewechselt wurde. Es ist offensichtlich, daß die Kennummer nur nach einem Wechsel des Datenträgers gelesen werden sollte. Wie erkennt OS/2 einen Wechsel? OS/2 kann dies nicht erkennen, wohl aber der Einheitentreiber. Falls die Einheit ein nicht austauschbares Medium erhält, muß die Datenträgerkennummer nicht überprüft werden. Der Einheitentreiber erkennt dies und antwortet mit *kein Wechsel*, wenn er nach dem Status des Mediums gefragt wird. Einige

austauschbare Mediumtreiber besitzen ein Flaggenbit, das beim Öffnen der Laufwerksklappe gesetzt wird. In diesem Fall antwortet der Einheitentreiber bei einer Anfrage mit *Wechselstatus unbestimmt*. Der Treiber erkennt nicht, ob der Datenträger tatsächlich ausgetauscht wurde, es könnte aber der Fall sein. In diesem Fall überprüft OS/2 die Datenträgerkennummer. Die erneute Überprüfung der Kennummer wird noch schwieriger, wenn eine Einheit, deren Datenträger austauschbar sind, keinen entsprechenden Indikator besitzt.

Der Entwickler eines Einheitentreibers muß einheitenspezifische Details beachten, um die minimale Zeitspanne zu bestimmen, in der ein Wechsel des Datenträgers auftreten kann. Ist die Einheit für Ein-/Ausgaben bereit und die minimal nötige Zeit zum Austausch des Datenträgers seit der letzten Operation noch nicht vergangen, erkennt der Treiber, daß noch derselbe Datenträger im Laufwerk liegen muß. Ist mehr Zeit vergangen, übergibt der Treiber eine *Mediumwechsel unbestimmt*-Meldung an OS/2 und das Betriebssystem muß den Datenträger überprüfen. Für Diskettenlaufwerke beträgt dieser Zeitintervall üblicherweise zwei Sekunden, so daß nach dem Verstreichen dieses Zeitraumes ein Extralesevorgang ausgeführt wird. Für schnell hintereinander erfolgende Diskettenein-/-ausgaben werden keine Extralesevorgänge benötigt.

Die Sicherstellung einer einzigartigen Datenträgerkennummer ist ein weiteres Problem. Das Betriebssystem darf sich nicht darauf verlassen, daß der Benutzer keine Datenträgernamen doppelt vergibt. Selbst ein "perfekter Benutzer" kann sich eine Diskette von seinem Nachbarn borgen, die einen Namen trägt, den der Benutzer schon einmal vergeben hat. OS/2 löst dieses Problem durch die Verwendung einer 32-bit-Zufallszahl als Diskettenkennummer. Bei der Formatierung einer Diskette gibt der Benutzer einen Namen ein. Von diesem Namen werden Prüfsummen gebildet und das Resultat, verknüpft mit der Anzahl der Sekunden zwischen dem aktuellen Zeitpunkt und dem Jahr 1980, wird als Ausgangswert für einen Zufallszahlengenerator verwendet. Der Zufallszahlengenerator gibt eine 32-bit-Zahl zurück, die als Datenträgerkennummer verwendet wird. Es ist

offensichtlich, daß die doppelte Vergabe einer Datenträgerkennummer zwar möglich ist, die vier Milliarden auf diese Weise zu erzeugenden Codes machen dies aber unwahrscheinlich.

Der vom Benutzer eingegebene Name dient lediglich dazu, in späteren Eingabeaufforderungen an den Benutzer klarzumachen, welche Diskette eingelegt werden soll. Der Name muß für das System nicht tatsächlich einzigartig sein. Ein Benutzer, der mehrere Disketten mit dem Namen *Arbeit* verwendet, wird unter OS/2 dadurch keine Probleme hervorrufen, da das Betriebssystem die Disketten aufgrund der Datenträgerkennummern als unterschiedlich erkennt. Das Einlegen der falschen Diskette mit Namen *Arbeit* als Reaktion auf eine Eingabeaufforderung wird von OS/2 abgelehnt und eine erneute Aufforderung zum Einlegen der Diskette ausgegeben. Nachdem das System mehrmals zum Einlegen der Diskette mit dem Namen *Arbeit* aufgefordert hat, wird sich der Benutzer aber sicherlich zur Umbenennung seiner Disketten entscheiden.

Das schwierigste Problem ergibt sich aus nicht benannten Disketten - dies sind Disketten, die unter MS-DOS formatiert wurden. Es kann auch hier nicht vom Benutzer erwartet werden, daß den Disketten ein einzigartiger Name zugewiesen wird. Gleichfalls ist die automatische Benennung durch OS/2 nicht vertretbar, da Datenträger *nur-lesbar* sein können. Auch wenn dies nicht der Fall ist, ergibt sich das Problem mit Disketten niedriger und hoher Dichte. Disketten niedriger Dichte können in einem Laufwerk für Disketten hoher Dichte gelesen werden, Schreibvorgänge eines mit hoher Dichte arbeitenden Laufwerkes auf eine Diskette mit niedriger Dichte können aber nur von Laufwerken mit hoher Dichte gelesen werden. Das Lesen der neuen Information durch Laufwerke niedriger Dichte ist nicht sichergestellt und der Startsektor ist bei der Verwendung eines Laufwerkes niedriger Dichte dann unter Umständen nicht mehr lesbar.

Für Datenträger ohne korrekte Datenträgerkennummer versucht OS/2 die Erstellung einer einzigartigen Substitutionsdatenträgerkennummer, indem Prüfsummen von Teilen des Stammver-

zeichnisses und der FAT-Tabelle gebildet werden. OS/2 verwendet dabei den Datenträgernamen (sofern existent). Ist kein Datenträgername vorhanden, nennt OS/2 die Datei *Diskette, die die Datei XXX enthält*, wobei XXX der Dateiname ist, der über eine Nummer geöffnet wurde. Keine der Methoden ist narrensicher und sie erfordern mehrere Diskettenzugriffsoperationen bei der Verifikation eines Datenträgers. Die beste Abhilfe besteht darin, Disketten, die in OS/2-Systemen zum Einsatz gebracht werden sollen, mit Namen zu versehen. OS/2-Namen sind abwärtskompatibel zu MS-DOS Version 3.X.

Der Befehl DISKCOPY erstellt unter OS/2 eine byteorientierte Kopie einer Diskette. Die einzige Abweichung der duplizierten Diskette vom Original besteht darin, daß diese eine andere Datenträgerkennummer im Startsektor aufweist (der Datenträgername wird nicht verändert). Der Benutzer von OS/2 erkennt dies nicht, da die DISKCOMP-Utility insofern Falschangaben macht, als das zwei Disketten als identisch angesehen werden, wenn sie bis auf die Datenträgerkennummer übereinstimmen. Wenn ein Benutzer eine Diskette mit DISKCOPY unter OS/2 dupliziert und Original und Kopie mit dem Befehl DISKCOMP unter MS-DOS 3.X vergleicht, wird ein Unterschied festgestellt.

Die Erläuterungen haben sich bis zu dieser Stelle auf Lese- und Schreibvorgänge über eine geöffnete Nummer beschränkt. Lese- und Schreibvorgänge sind *datenträgerorientierte* Operationen, da sie sich immer auf den Datenträger beziehen, auf dem die Datei vorliegt. Der *DosOpen*-Aufruf ist dagegen *laufwerksorientiert*, da das Standard- oder das spezifizierte Laufwerk nach der entsprechenden Datei durchsucht wird, wobei es gleichgültig ist, welcher Datenträger im Laufwerk liegt. Alle Nummernoperationen sind laufwerksorientiert, alle Aufrufe auf Namensbasis sind datenträgerorientiert. Derzeit kann nicht spezifiziert werden, daß eine Datei auf einem bestimmten Datenträger angelegt oder geöffnet wird. Um sicherzustellen, daß eine Zwischen- oder Ausgabedatei auf einem bestimmten Datenträger angelegt wird, muß die Datei auf diesem Datenträger geöffnet sein. Dazu wird direkt vor dem Öffnen ein Schreibvorgang in die Datei ausgeführt. Die Schreiboperation gefolgt von einem *DosBufReset*-

Aufruf stellt sicher, daß der gewünschte Datenträger im Laufwerk einliegt.

15.3 Effizienz bei der Ein- und Ausgabe

OS/2 unterstützt blockorientierte Operationen (zerlegend oder nicht zerlegend) bei allen Diskettenein-/-ausgabeoperationen. Ein Programm kann eine beliebige Anzahl von Bytes lesen oder schreiben. OS/2 liest die entsprechenden Sektoren und internen Puffer, so daß nur die betroffenen Bytes geladen werden. Jeder *DosRead*- und *DosWrite*-Aufruf benötigt Ausführungszeit, daher ist es effektiver, mit einem Aufruf eine größere Datenmenge anzusprechen.

Der Durchsatz bei der Ein-/Ausgabe wird nochmals verbessert, wenn eine Sektorgrenzanpassung ausgeführt wird. Das Lesen eines ganzzahligen Vielfachen von 512 Bytes ab einer Sektorengrenze führt dazu, daß vollständige Sektoren ohne Zwischenspeicherung in Systempuffern an die Anzeigehardware übergeben werden. Da das Dateisystem auf eine optimale Speicherung der Daten auf dem Datenträger achtet (logisch zusammenhängende Daten werden so gespeichert, daß das Lesen oder Schreiben ohne umfangreiche Änderung der Schreib-/Lesekopfposition erfolgt), ist das Lesen von vier aufeinanderfolgenden Sektoren (2048 Bytes) nur unwesentlich langsamer als das Lesen eines Sektors (512 Bytes).

Wenn die Länge oder Dateiposition einer Anforderung kein Vielfaches von 512 Bytes ist, führt OS/2 nur die Zerlegung unvollständig zu lesender Sektoren über die Puffer aus. Vollständig zu übertragende Sektoren werden direkt übergeben. Die Übertragung großer Datenmengen mit einer Operation ist auch dann sinnvoll, wenn die Sektorengrenzen bei Anforderungen nicht berücksichtigt werden.

Es läßt sich also feststellen, daß die Effizienz bei der Ein- und Ausgabe am größten ist, wenn umfangreiche Datenmengen angesprochen werden und die Datenanforderungen an den Sektorengrenzen ausgerichtet sind. Auch nicht ausgerichtete Datenan-

forderungen können mit einem guten Durchsatz ausgeführt werden, wenn umfangreiche Datenmengen in einer Operation bearbeitet werden. Bei Programmen, die keine dieser beiden Voraussetzungen erfüllen, empfiehlt sich die Verwendung der Routinen zum "Blocken" und "Entblocken". Dies gilt auch bei umfangreichen Datenanforderungen, die nicht an Sektorengrenzen ausgerichtet sind. Programme mit häufigen kleinen Datenanforderungen ohne Sektorenausrichtung arbeiten am besten, wenn Sektorenblöcke in interne Puffer gelesen und anschließend vom Programm selbst "entblockt" werden. Dies spart die Zeit für häufig auftretende *DosRead*- und *DosWrite*-Aufrufe.

Kapitel 16

Überwachung, Datenintegrität, Zeitgeberroutinen

Bei der Erläuterung der Entwicklungsziele von OS/2 wurde die Unterstützung der MS-DOS-Funktionen erwähnt, selbst wenn dazu die Umgehung des Betriebssystems nötig wird. Ein gutes Beispiel hierfür stellt die Manipulation von Daten dar, die von externen Einheiten kommen, eine Programmiertechnik, die üblicherweise das Verbiegen von Interruptvektoren voraussetzt und die von vielen Anwendungsprogrammen eingesetzt wird. Damit sind speicherresidente Programme wie beispielsweise *SideKick* populär geworden. Diese Programme werden in den Speicher geladen und verbleiben dort (auch nach der Beendigung). Beim ersten Aufruf wird der Tastaturinterruptvektor auf den Code des speicherresidenten Programms gerichtet. Die Programme untersuchen jeden Tastenanschlag, um festzustellen, ob die Aktivierungstaste betätigt wurde. Ist dies nicht der Fall, wird die Kontrolle an die ursprüngliche interruptbearbeitende Routine weitergegeben. Falls der Tastenanschlag die aktivierende Tastenfolge ist, verbleibt die Kontrolle beim speicherresidenten Programm und eine entsprechende Meldung oder ein Menü erscheint auf dem Bildschirm. Andere Programme verbiegen den Tastaturvektor, um Tastaturerweiterungsmakros zu ermöglichen. Auch der BIOS-Vektor für den Drucker wird oft verändert, um andere Druckertreiber einzusetzen oder die an den Drucker gesendeten Daten zu manipulieren.

Im allgemeinen werden die Interruptvektoren verändert bzw. "verbogen", die Einheiteninterrupts entsprechen und mit Einheitentreiberroutinen im ROM-BIOS kommunizieren. Die weite

Verbreitung der speicherresidenten Programme zeigt das Verlangen der Anwender nach Programmen, die Datenströme aus externen Einheiten überwachen bzw. modifizieren. Der OS/2-Mechanismus zur Ersetzung speicherresidenter Programme sind Einheitenüberwachungsroutinen (*device monitor*).

16.1 Kontrolle der Geräteeinheiten

Das Konzept der Einheitenüberwachungsroutinen mußte den allgemeinen Anforderungen genügen, die an OS/2 zu stellen waren. Insbesondere die MS-DOS-Methode der Umleitung von Interrupts an Anwendungen, indem Interruptvektoren entsprechend verändert werden, konnte in einem Multitasking-Betriebssystem wegen der Grundsatzforderung einer stabilen Umgebung nicht erlaubt werden. Hinzu kommt, daß OS/2 (im Gegensatz zu MS-DOS) das ROM-BIOS nicht als eine Art Einheitentreiber ansieht, so daß das Verbiegen der BIOS-Kommunikationsvektoren kein Zugriff auf Datenströme aus Einheiten bereitstellen kann. Die willkürliche Veränderung der Operation eines Einheitentreibers paßt nicht zum Grundkonzept von OS/2. Der Einheitentreiber ist das Architekturelement, das die Informationen über die Einheit enthält, er muß daher in jede Veränderung der Datenströme einbezogen sein. Das Resultat ist ein OS/2-Einheitenüberwachungssystem, das den Prozessen die Ausführung normaler Ring-3-Befehle (Benutzermodus) zum Überwachen und Verändern von Datenströmen von und zu Geräteeinheiten erlaubt (immer mit Kenntnis des entsprechenden Einheitentreibers).

Ein Prozeß vergibt für sich selbst den Status als Einheitenüberwachungsprozeß, indem der entsprechende Einheitentreiber mit einem *DosMonReq*-Aufruf ausgeführt wird (ein Dynlink-Paket, das gegebenenfalls den Einheitentreiber über einen *DosDevIOCtl*-Aufruf einbezieht). Der Prozeß muß zwei Datenpuffer bereitstellen, einen für ankommende und einen zweiten für ausgehende Überwachungsdaten. Prozesse können OS/2 aufrufen, OS/2 selbst hat aber keine Möglichkeit zum Aufruf eines Prozesses. Der Signalmechanismus macht hier eine Ausnahme (wo-

bei aber die erläuterten Einschränkungen zu berücksichtigen sind). Unter OS/2 können diese Beschränkungen umgangen werden, indem die normale Definition der Aufruf-/Rückkehrsequenz umgekehrt wird. Soll OS/2 einen Prozeß aufrufen, muß zuerst der Prozeß einen Aufruf an OS/2 mit einem seiner Threads ausführen. OS/2 hält diesen Thread beschäftigt, bis das *Rückrufereignis* auftritt. OS/2 simuliert den Aufruf eines Prozesses durch die Freigabe des Threads, so daß dieser vom Systemaufruf zurückkehrt und die weitere Prozeßbearbeitung aufgenommen werden kann. Wenn der Prozeß zur Rückkehr zu OS/2 bereit ist, wird der Einsprungspunkt wieder aufgerufen (siehe Bild 16.1).

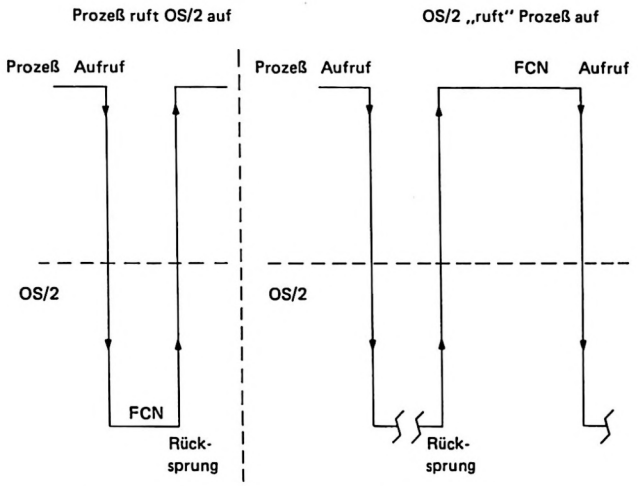


Bild 16.1 "Aufruf" eines Prozesses von OS/2

OS/2 verwendet diese Programmiertechnik auch zur Überwachung. Ein Überwachungsprozeß muß den OS/2-Einsprungspunkt aufrufen, nachdem er sich als Geräteüberwachungsprozeß angemeldet hat. OS/2 meldet dem Überwachungsprozeß das Vorhandensein von Daten im Puffer für ankommende Daten

und gestattet dem Thread die Rückkehr zum Prozeß. In Bild 16.2 wird eine Einheit mit den zwei Überwachungsprozessen X und Y gezeigt.

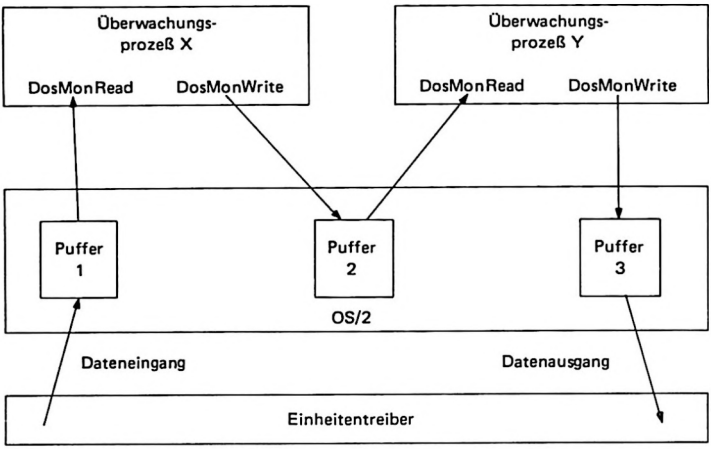


Bild 16.2 Einheitenüberwachungsprozesse

Einige weitere Details verdienen Beachtung. OS/2 muß für jeden Prozeß eine konsistente Umgebung bereitstellen, wobei kein anderer Prozeß ein Störfaktor werden darf. Jede Einheit kann so viele Überwachungsprozesse besitzen, wie der Einheitentreiber erlaubt. OS/2 verknüpft mehrere Einheitenmonitore zu einer Monitorkette, so daß der Datenstrom durch den ersten Monitor der Kette, dann durch den zweiten Monitor usw. übertragen wird. Bei der Anmeldung eines Prozesses als Überwachungsprozeß muß spezifiziert werden, ob dies der erste oder letzte Überwachungsprozeß in der Kette sein soll. Für einige Anwendungen ist die Reihenfolge der Monitorprozesse (Überwachungsprozesse) von Bedeutung. Der Monitor, der sich als erster in der Kette anmeldet, ist auch tatsächlich der erste Überwachungsprozeß; der nächste Überwachungsprozeß, der den ersten Platz in der Kette anfordert, wird zum zweiten Prozeß usw. Dasselbe Schema gilt auch für Überwachungsprozesse, die

am Ende der Kette stehen sollen. Der erste Prozeß, der sich für den letzten Platz in der Kette anmeldet, wird tatsächlich der letzte Prozeß; der nächste sich als letzter in der Kette anmeldende Prozeß wird zum vorletzten Prozeß usw.

Das Format des überwachten Datenstromes ist einheitenspezifisch und wird vom Einheitentreiber festgelegt. Einige Einheitentreiber befolgen Sonderregeln; so erlaubt beispielsweise der Tastaturtreiber den Überwachungsprozessen das Einfügen eines Bildschirmumschaltungssignales in den Datenstrom, woraufhin das Signal bearbeitet wird, als wäre es über die Tastatur eingegeben worden. Der Einheitentreiber übergibt diese Tastenfolge (bei einer Eingabe über die Tastatur) aber nicht durch die Überwachungsprozeßkette, sondern direkt.

Dies bietet einen Schutz vor fehlerhaften Tastaturüberwachungsprozessen, die andernfalls das System zum Absturz bringen können, indem alle Versuche des Benutzers zur Bildschirmumschaltung unterdrückt werden. Der Einheitentreiber für den Bildschirm erlaubt nicht die Verwendung von Einheitenüberwachungsprozessen, da das VIO-Paket und der Presentation Manager den Bildschirmeinheitentreiber umgehen, so daß dieser keine an den Bildschirm gerichteten Daten empfängt. Hinzu kommt (was aber nicht der eigentliche Grund ist), daß eine Verringerung des Datendurchsatzes eintreten würde.

Der Thread wird durch einen *DosMonRead*-Aufruf beschäftigt gehalten, bis ankommende Daten verfügbar sind. Der *DosMonWrite*-Aufruf gibt die CPU sobald als möglich an den Prozeß zurück. Der *DosMonRead* aufrufende Thread muß nicht der Thread sein, der den *DosMonWrite*-Aufruf ausführt.

Da Überwachungsprozesse eine wichtige Komponente von OS/2 darstellen, müssen sie von Anwendungsprogrammen mit Bedacht eingesetzt werden. Zunächst einmal werden Überwachungsprozesse in eine Einheitenkette eingereiht, was natürlich Auswirkungen auf den Datendurchsatz der Einheit hat. Bei der Betätigung einer Taste durch den Benutzer muß beispielsweise ein Datenpaket durch jeden Überwachungsprozeß in der Tastaturkette übergeben werden, bevor die Anwendung den Tastenan-

schlag bearbeiten kann. Durch den nicht optimierten Einsatz von Überwachungsprozessen oder durch zu viele Überwachungsprozesse in einer Kette werden die Antwortzeit des Systems merklich herabgesetzt. Dem Thread zum Lesen, Bearbeiten und Schreiben der Überwachungsdaten sollte eine hohe Priorität zugeordnet werden, empfohlen wird die niedrigste Prioritätskategorie für eine erzwungene Bearbeitung. Hinzu kommt, daß die Überwachungskomponente einer überwachenden Anwendung keine kritischen Bereiche enthalten darf. Gleiches gilt für andere Ereignisse, die eine Operation verlangsamen oder unterdrücken können. Es ist zu berücksichtigen, daß bei sehr ausführlichen Überwachungsdatenströmen ein Thread mit normaler Priorität zur Bearbeitung der Daten verwendet werden muß, damit der Thread höherer Priorität die Übertragung von Überwachungsdaten in beide Richtungen weiterführen kann. So müssen beispielsweise ein Hilfsthread und ein Puffer verwendet werden, wenn ein Tastaturmonitor alle Tastenanschläge in einen Disketten-/Plattenpuffer schreiben soll.

Schließlich ist noch zu berücksichtigen, daß die Daten in einem Überwachungspuffer verloren sind, wenn der Überwachungsprozeß unerwartet beendet wird (obwohl OS/2 den Prozeß korrekt aus der Monitorkette herausnimmt). Es liegt auf der Hand, daß der Verlust einer nicht bekannten Datenmenge aus dem Tastatureingabestrom oder bei der Druckerausgabe zu Problemen führt. Überwachungsprozesse müssen folglich mit Bedacht entwickelt werden, um diese Risiken zu minimieren.

Überwachungsprozesse sind mehr als andere Leistungsmerkmale geeignet, die grundliegenden Architekturprinzipien von OS/2 zu verletzen. Das Vorhandensein von Überwachungsprozessen im System zeigt, wie wichtig dieses Leistungsmerkmal ist. Insbesondere wird durch Überwachungsprozesse die Konzeption der minimalen Einflußnahme zwischen Prozessen (Vereinzelung der Prozesse) verletzt. Es ist offensichtlich, daß ein Prozeß bei der Überwachung von Datenströmen die Aus- oder Eingabe vieler Prozesse beeinflussen kann. Teilweise wird dies als Leistungsmerkmal und nicht als Fehler angesehen. So dient beispielsweise der Druckerspooler der Überwachung der Druckerausgabe. Die

Daten sollen auf Diskette oder Festplatte gespeichert und von der Datei an den Drucker übergeben werden, sobald dieser betriebsbereit ist. Die Zwischenspeicherung von Druckausgaben ist hier eine erwünschte Einmischung in einen anderen Prozeß. Die Entwickler von Überwachungsanwendungen müssen darauf achten, daß ihre Anwendungen weder den Datendurchsatz des Systems noch dessen Stabilität angreifen.

16.2 Datenintegrität

Die Datenintegrität in Multitasking-Systemen wurde bereits mehrmals angesprochen. In diesem Abschnitt soll das bisher Erläuterte nicht wiederholt werden, sondern es sollen die wichtigsten Elemente und einige damit verwandte Systemleistungsmerkmale im Zusammenhang dargestellt werden.

Das erste Problem in einem Multitasking-System ist, daß mehrere Prozesse (oder mehrere Threads in einem Prozeß) gleichzeitig auf die gleichen Ressourcen zugreifen können - auf eine Datei, eine Geräteeinheit, eine Datenstruktur im Speicher oder sogar nur auf ein einzelnes Byte im Hauptspeicher. Wenn die Manipulation von Ressourcen durch verschiedene Elemente nur nacheinander ausgeführt werden kann, wird dies *kritischer Bereich* genannt. Der Ausdruck bezieht sich auf die Manipulation der Ressourcen, nicht unbedingt auf den ausführenden Progammcode. Es ist klar, daß jede von vier Unterroutinen ein kritischer Bereich ist, wenn diese bestimmte Ressourcen (einen kritischen Bereich) verändern.

Das Problem ist komplexer als ein Programmierer annehmen könnte, der mit Multitasking-Systemen nicht vertraut ist. So kann beispielsweise das Worttesten und die dazugehörige Inkrementierung zu einem Problem werden, falls es im kritischen Bereich liegt. Sofern jeweils nur ein Thread in einem Prozeß auf das Wort zugreifen kann, ist die Hintereinanderschaltung der Zugriffe auf den kritischen Bereich sichergestellt. Besitzt aber mehr als ein Thread Zugriff auf das Wort, so können mehrere gleichzeitig ausgeführte Threads im kritischen Bereich zu Kollisionsproblemen führen. Betrachten Sie die Assemblerbe-

fehlsfolge in Listing 16.1. Die Aufgabe dient der Prüfung der Variablen ZAEHL (ist der Wert 4 gespeichert?); fällt der Test negativ aus, wird ZAEHL inkrementiert. Falls der Wert 4 enthalten ist, wird zur Marke FERTIG gesprungen. In Listing 16.2 wird gezeigt, was in einer Umgebung passieren kann, die aus mehreren Threads besteht: Thread A prüft den Wert, um festzustellen, ob der Wert 4 in ZAEHL enthalten ist. Das Ergebnis besagt, daß der Wert 3 gespeichert ist. Sofort nach der Vergleichsanweisung erfolgt im System eine Kontextumschaltung und Thread B wird ausgeführt. Thread B vollführt ebenfalls einen Vergleich und erkennt, daß der Wert 3 gespeichert ist. Die Variable wird von Thread B inkrementiert. Nachdem Thread A die Ausführung wieder aufgenommen hat (hinter der Vergleichsanweisung) glaubt Thread A, daß der Wert von ZAEHL noch 3 beträgt. Der Wert von ZAEHL wird nochmals inkrementiert (nun von Thread A). Der tatsächliche Wert von ZAEHL ist damit 5 und über die Grenze von 4 (was als obere Begrenzung für den Wert der Variable ZAEHL festgelegt wurde) inkrementiert worden. Die Marke FERTIG wird niemals erreicht.

```
ZAEHL    DW   0    ; Ereigniszähler
     .
     .
     .
     CMP      ZAEHL,4     ; Wert 4 gespeichert?
     JE       FERTIG      ; Ja, dann ok
     INC      ZAEHL       ; Inkrementierung
     .
     .
     .
```

Listing 16.1

```
Thread A                                Thread B
    .
    .
    CMP     ZAEHL,4  [ZAEHL besitzt den Wert 3]
    --------------------------Kontextumschaltung------->
                                CMP     ZAEHL,4 [ZAEHL besitzt den Wert 3]
                                JE      FERTIG
                                INC     ZAEHL   [ZAEHL besitzt den Wert 4]
    <-------Kontextumschaltung--------------------------
    JE      FERTIG  [Sprung wird nicht ausgeführt]
    INC     ZAEHL   [ZAEHL besitzt nun den Wert 5]
```

Listing 16.2

Der nur wenig mit Multitasking-Umgebungen vertraute Programmierer könnte nun einwenden, daß das Auftreten dieser Situation unwahrscheinlich ist. Vielleicht stehen die Chancen Eins zu einer Million, daß diese Befehlsfolge mit genau diesen Kontextumschaltungen ausgeführt wird. Da ein Mikroprozessor aber eine Million Anweisungen in der Sekunde ausführt, wird es nicht lange dauern, bis dieser unglückliche Fall eintritt. Hinzu kommt, daß ein inkorrektes Programm üblicherweise mehrere nicht geschützte kritische Bereiche besitzt, die häufig länger als zwei Anweisungen (wie im einfachen Beispiel) sind.

Ein unter OS/2 ablauffähiges Programm muß alle kritischen Bereiche erkennen und schützen. Ein Programm, das dazu nicht in der Lage ist, wird nicht einwandfrei funktionieren. Der Programmierer darf sich nicht darauf verlassen, daß OS/2 keine Kontextumschaltung im kritischen Bereich durchführt. OS/2 kann zu jeder Zeit Umschaltungen vornehmen. Da Umschaltungen durch nicht vorhersehbare externe Ereignisse hervorgerufen werden können (beispielsweise Ein-/Ausgaben am seriellen Port oder Diskettenoperationen), kann auch durch das Austesten eines Programmes nicht sichergestellt werden, daß ein kritischer Bereich geschützt ist. In der Praxis sind Testumgebungen häufig recht einfach: Kontextumschaltungen finden dort in gleichblei-

benden Zeitintervallen statt und die bisher genannten Probleme treten während der Testläufe nur selten auf. Im tatsächlichen Einsatz hingegen wird ein solches Programmsystem sehr schnell instabil.

Selbstverständlich ist das Austesten eines Programms sehr wichtig; die einzig sichere Möglichkeit zur Behandlung kritischer Bereiche ist die sorgfältige Untersuchung des Programms unter der Annahme, daß alle Threads im System gleichzeitig ausgeführt werden. Dies ist mehr als ein Gedankenexperiment. Es werden in naher Zukunft Computersysteme mit mehreren Prozessoren entwickelt werden, so daß dann unter OS/2 mehrere Threads tatsächlich gleichzeitig ausgeführt werden. Bei der Untersuchung des Programmes sollte auch angenommen werden, daß die CPU immer im denkbar ungünstigsten Fall eine Kontextumschaltung vornimmt. Das Auftreten der Umschaltung an einer vom Programmierer nicht berücksichtigten Stelle wird beim Einsatz des Programmes bald in Erscheinung treten.

Solche Probleme werden beim Austesten eines Programmes häufig nicht festgestellt; bei komplexen Systemen sind diese Fehler nur schwer zu finden. Ein Programmierer sollte also keine Mühe scheuen, alle kritischen Bereiche eines Progamms im vorhinein zu erkennen.

16.2.1 Semaphoren

Die herkömmliche Lösung zum Schutz eines kritischen Bereichs ist die Semaphore. Die zwei unter OS/2 zur Verfügung stehenden Arten von Semaphoren - RAM- und Systemsemaphoren - haben beide ihre Vorteile, der Schutz der kritischen Bereiche ist garantiert. Die Operation einer Semaphore ist ein nicht teilbarer Prozeß. Wenn ein Thread kritische Ressourcen manipulieren will, muß er erst eine Semaphore zum Schutz der Ressourcen anfordern. Erst nach Erhalt der Semaphore darf mit dem Wert der Ressourcen gearbeitet werden, da dieser zwischen der Anforderung der Semaphore und der Zuweisung der Semaphore verändert worden sein könnte. Nach der Vervollständigung der

Ressourcenmanipulation wird die Semaphore wieder freigegeben.

Der Semaphorenmechanismus schützt auch gegen kooperierende Threads, unabhängig davon, ob diese zum selben oder zu unterschiedlichen Prozessen gehören. Ein weiterer OS/2-Mechanismus (*DosEnterCritSec*) steht zum Schutz eines kritischen Bereiches zur Verfügung, auf den nur Threads eines einzelnen Prozesses zugreifen. Wenn ein Thread einen *DosEnterCritSec*-Aufruf ausführt, unterdrückt OS/2 die Ausführung aller anderen Threads dieses Prozesses bis zur Beendigung der Abarbeitung eines *DosEnterCritSec*-Aufrufs. Natürlich werden nur Threads im Anwendungsmodus unterdrückt. Im OS/2-Kernel ausgeführte Threads werden solange nicht beeinflußt, bis sie wieder in den Anwendungsmodus eintreten.

Die Verwendung von *DosEnterCritSec* ist nicht ungefährlich, da die anderen Threads des Prozesses unterbrochen werden können, während sie in kritischen Bereichen beschäftigt sind. Falls der den *DosEnterCritSec*-Aufruf ausführende Thread den kritischen Bereich ebenfalls bearbeiten will, führt dies für den Prozeß zu einem *Deadlock*. Ist ein Dynlink-Paket in die Bearbeitung einbezogen, kann dieses eigene Threads angelegt haben, die dem Klientenprozeß nicht bekannt sind. Der Klientenprozeß hat möglicherweise noch nicht einmal Kenntnis von der Existenz eines kritischen Bereiches. Daher ist der *DosEnterCritSec*-Aufruf nur sicher, wenn er dem Schutz kleiner Programmteile dient, die nicht zu einer Blockierung oder einem *Deadlock* führen können. Zudem dürfen keine Dynlink-Module aufgerufen werden.

Hinweis: Es soll dem Leser als Aufgabe überlassen bleiben, herauszufinden, warum ein DosEnterCritSec-Aufruf nicht sicher ist, sofern nicht alle anderen Threads im Prozeß für diesen kritischen Bereich ebenfalls davon Gebrauch machen.

Ein anderes Leistungsmerkmal von OS/2 im Zusammenhang mit kritischen Bereichen ist die gemeinsame Verwendung von Dateien in Verbindung mit der Datensatzsperre zum Schützen kritischer Bereiche, die aus Dateien oder Teilen von Dateien bestehen. So betrachtet ein Datenbankprogramm eine Hauptdatenbankdatei sicherlich als kritischen Bereich und erlaubt keine

Veränderung, solange sie von der Anwendung geöffnet ist. Die Datei kann in einem Modus geöffnet werden, in dem keine anderen Zugriffe auf die Datei zugelassen werden. Solange die Datenbankanwendung die Datei geöffnet hält, wird von OS/2 jeder Versuch des Öffnens (oder Löschens) der Datei durch andere Prozesse unterbunden.

Die Datensatzsperre kann zum Schutz kleinerer Dateneinheiten verwendet werden. Ein Prozeß kann einen Bereich mit mehreren Bytes in einer Datei sperren, und OS/2 hindert alle anderen Prozesse am Lesen und Schreiben der Bytes, solange die Sperre aktiv ist. Die beiden Sonderformen des Schutzes kritischer Bereiche sind insofern einzigartig, als sie einen Prozeß gegen alle anderen Prozesse schützen, was auch für Prozesse gilt, die ihren eigenen Zugriff auf kritische Bereiche nicht schützen. Leider bieten die Mechanismen der gemeinsamen Verwendung von Dateien und der Datensatzsperre keine Möglichkeit zur Blockierung, bis dieser Konflikt gelöst ist. Anwendungen, die auf die Konfliktlösung warten, müssen eine Polling-Schleife verwenden. Der Anwendungsprogrammierer sollte einen *DosSleep*-Aufruf zum Blockieren für mindestens eine halbe Sekunde nach jedem Polling-Durchlauf einsetzen.

Obwohl Semaphoren kritische Bereiche schützen, werfen sie selbst Probleme auf. Was passiert beispielsweise, wenn ein asynchrones Ereignis (zum Beispiel eine Programmbeendigung oder ein Signal) die CPU bei der Bearbeitung eines kritischen Bereiches unterbricht und nicht zurückkehrt, um die Semaphore freizugeben? Die Möglichkeiten sind vielfältig; einige sind im folgenden aufgeführt:

- *Ignorieren.* Dies ist möglich, solange der kritische Bereich einem einzigen Prozeß angehört und der Prozeß keine Signale zur Modifikation der Ausführungsfolge verwendet. Zudem dürfen weder der Prozeß noch die davon aufgerufenen Dynlink-Routinen den Zugriff auf den kritischen Bereich während der *DosExitList*-Verarbeitung versuchen.
- *Freigabe der Semaphoren.* Dies ist eine Möglichkeit, die offen steht, solange die durch die Semaphore geschützten

Ressourcen keinen Status besitzen (wie eine Semaphore, die das Recht des Schreibens auf den Bildschirm schützt). Der Trick besteht darin, dafür zu sorgen, daß tatsächlich der unterbrochene Thread die Semaphore gesetzt hat. Die Semaphore darf nicht freigegeben werden, wenn sie nicht von diesem Thread gesetzt wurde. Falls eine Semaphore nur in einem einzigen Prozeß verwendet wird, können *DosExitList*-bearbeitende Routinen die Freigabe der Semaphore erzwingen.

- *Erkennen der Situation und Wiederherstellen des kritischen Bereiches.* Die Prüfung kann nur für RAM-Semaphoren ausgeführt werden, wenn dies während der Prozeßbeendigung stattfindet und die Semaphore nur von diesem Prozeß verwendet wird. In diesem Fall ist bekannt, daß ein Thread des Prozesses die Semaphore gesetzt hat. Gleichermaßen ist bekannt, daß dieser Thread den kritischen Bereich nicht mehr bearbeitet, da alle Threads beendet sind. Die Semaphore kann durch die Verwendung eines nicht blockierenden *DosSemSet*-Aufrufs getestet werden. Bei gesetzter Semaphore müssen die Ressourcen bereinigt werden.

  Systemsemaphoren eignen sich zu diesem Zweck besser. Bei der Beendigung des Besitzer-Threads einer Systemsemaphore wird die Semaphore markiert. Der nächste Versuch des Setzens der Semaphore liefert einen Code, der dem neuen Besitzer die Beendigung des vorigen Besitzer im kritischen Bereich mitteilt. Der neue Besitzer kann die Ressourcen freigeben.

Eine weitere Möglichkeit ist der Versuch, die CPU am Verzweigen während der Bearbeitung eines kritischen Bereiches zu hindern. Signale können mit dem *DosHoldSignal*-Mechanismus verzögert werden. Die von einem externen Aufruf initiierte Prozeßbeendigung kann durch die Initialisierung einer signalbearbeitenden Routine für das KILL-Signal und der Verwendung des *DosHoldSignal*-Aufrufs verzögert werden. Dies schützt aber nicht vor einer Programmbeendigung aufgrund allgemeiner Fehler.

16.2.2 DosBufReset

Die Synchronisation von Disketten-/Plattendaten - ebenfalls ein Aspekt der Datenintegrität - hat nichts mit kritischen Bereichen zu tun. Häufig schreibt OS/2 die für die Disketten-/Plattenübertragung bestimmten Daten aufgrund eines *DosWrite*-Aufrufs nicht direkt auf Diskette/Platte, sondern speichert sie in einem Puffer. Nachfolgende Leseaufrufe auf die Daten werden korrekt ausgeführt, so daß eine Anwendung nicht erkennt, ob die Daten bereits auf Diskette/Platte oder nur in einem Puffer im Hauptspeicher vorliegen. Dies kann festgestellt werden, wenn direkte physikalische Adressen des Datenträgers durch ungefilterte Datenträgerlesevorgänge untersucht werden. Dies erklärt, warum CHKDSK keinen Datenträger prüft, auf dem geöffnete Dateien vorliegen; die Meldung nicht existierender Fehler wäre möglich.

OS/2 schreibt die Daten zu einem vom Betriebssystem bestimmten Zeitpunkt auf die Diskette, so daß die Pufferung nur bei einem Systemabsturz in Betracht gezogen werden muß, bei dem gepufferte, aber noch nicht geschriebene Daten im Hauptspeicher vorliegen. Diese Art des Systemabsturzes ist selten, für einige Anwendungen kann dies aber bedrohlich sein, so daß ein Schutz erforderlich ist. Die drei hierfür vorgesehenen OS/2-Funktionen sind datenträgerorientiertes Löschen (*volume flushing*), nummernorientiertes Löschen (*handle flushing*) und Durchschreiben (*writethrough*). Beide Löschoperationen schreiben alle Puffer - mit den veränderten, aber noch nicht auf Diskette/Platte geschriebenen Daten - auf den Datenträger. Nach der Rückkehr vom Aufruf befinden sich die Daten auf der Diskette bzw. Platte. Man sollte den Aufruf aber nur selten einsetzen, da er häufig zur Folge hat, daß alle Pufferinhalte im System auf den Datenträger übertragen werden. Falls zu einem Server-Gerät geöffnete Dateinummern in einem Netzwerk vorhanden sind, werden die meisten oder alle Puffer des Servers ebenfalls gelöscht. Auch bei der nummernorientierten Operation kann das Löschen der Puffer im Netzwerkserver ausgelöst werden. Dabei ist zu beachten, daß eine Löschoperation der Puffer nicht nur die Daten zu einem früheren Zeitpunkt auf den Da-

tenträger schreibt, sondern auch weitere Schreibvorgänge erfordert. Ein Beispiel hier ist eine Anwendung, die bei jedem Schreibvorgang zehn Bytes überträgt. In diesem Fall puffert OS/2 die Daten, bis ein Sektor mit Daten gefüllt ist. Eine Reihe von Pufferlöschoperationen würde zu diesem Zeitpunkt den Sammelpuffer mehrmals (unnötigerweise) auf den Datenträger schreiben.

16.2.3 Durchschreiben

Pufferlöschungen sind kostenintensiv. Zudem garantieren sie keine bestimmte Reihenfolge beim Schreiben der Daten, solange sie nicht häufig verwendet werden. Einige Anwendungen (wie Datenbankverwaltungssysteme) müssen dafür Sorge tragen, daß auf den Datenträger geschriebene Daten in derselben Reihenfolge vorliegen, wie dies mit einem *DosWrite*-Aufruf unter OS/2 der Fall wäre. So kann es für eine Anwendung wichtig sein, daß die eigentlichen Daten vor der Belegungskette auf den Datenträger geschrieben werden und erst nach der Übertragung der Belegungskette das Datenbankverzeichnis aktualisiert wird. Diese Reihenfolge kann die Wiedergewinnung von Daten im Fehlerfall für das Datenbankverwaltungssystem vereinfachen.

Der hier verwendete OS/2-Mechanismus ist das *Durchschreiben* (*writethrough*) - dafür wird ein Statusbit verwendet, das für den gesamten Prozeß oder für einzelne Dateinummern gesetzt werden kann. Falls das Durchschreiben für den gesamten Prozeß oder für eine Nummer, für die ein Schreibvorgang ausgeführt wird, aktiv ist, schreibt OS/2 die Daten auf den Datenträger, bevor die *DosWrite*-Operation zurückkehrt. Es ist offensichtlich, daß Anwendungen unter Verwendung des Durchschreibens Daten in großen Mengen schreiben sollten. Das Schreiben kleiner Datenmengen ist hier sehr ineffizient.

Zwei Punkte sind bei der Verwendung des Befehls zum Durchschreiben zu beachten:

- Wird das Durchschreiben-Bit bei einer Datei nach dem Öffnen gesetzt, gilt dies für alle nachfolgenden Schreibvor-

gänge. Daten von vorherigen Schreibvorgängen können immer noch in Puffern vorliegen.

- Falls eine Durchschreibedatei von mehreren Prozessen verwendet werden wird oder über mehrere Nummern geöffnet ist, sollten alle Instanzen der Datei zum Durchschreiben markiert werden. Daten, die in eine zum Durchschreiben nicht geöffnete Nummer geschrieben werden, könnten andernfalls unter Umständen in Puffern abgelegt werden.

- Die Operationen des Pufferlöschens und Durchschreibens von Daten führen zu Überraschungen bei der Benutzung mit dem aktuellen FAT-Dateiverwaltungssystem. Obwohl diese Leistungsmerkmale die Daten auf Diskette/Platte schreiben, wird von ihnen der Verzeichniseintrag nicht auf die neue Länge der Datei gesetzt. Wird eine Datei um zehn Sektoren erweitert und tritt vor dem Schließen ein Systemabsturz auf, sind die letzten zehn Sektoren verloren. Trotz des Durchschreibens von Daten oder der Löschung von Puffern und obwohl die zehn Sektoren tatsächlich auf die Diskette geschrieben wurden, trägt CHKDSK die Sektoren in die Liste freier Sektoren ein, da der Verzeichniseintrag nicht aktualisiert wurde.

  Die Durchschreibeoperation schützt die Daten einer Datei, nicht aber Verzeichnis- oder Belegungsinformationen. Dies muß nicht berücksichtigt werden, solange ein Teil der Datei überschrieben wird, der gemäß Belegungstabelle und Verzeichniseintrag zur Datei gehört. Das Schreiben von Bytes, die die Dateilänge vergrößern, ist dagegen nicht geschützt. Hier werden zwar die Daten auf Diskette übertragen, das Verzeichnis aber nicht aktualisiert, und bei einem Systemabsturz kann der Teil der Datei, der eine Erweiterung darstellt, nicht gerettet werden. Die beste Lösung ist die Verwendung der Funktion *DosNewSize* zur Dateivergrößerung. Nachfolgend muß ein *DosBufReset*-Aufruf erfolgen, um die Daten auf den Datenträger zu übertragen. Die Erweiterung einer Datei sollte in ausreichendem Maße erfolgen, um mehreren Schreiboperationen gerecht zu wer-

den. Dies ist der wiederholten Verwendung der Kombination *DosNewSize/DosBufReset* vorzuziehen. Falls am Ende festgestellt werden sollte, daß zuviel Speicherplatz für die Datei belegt wurde, kann vor dem Schließen ein *DosNewSize*-Aufruf zur Anpassung der Dateilänge ausgeführt werden.

16.3 Zeitgeberroutinen

Echtzeitanwendungen sind keine Seltenheit. Ein Spieleprogramm kann beispielsweise asynchrone Ereignisse abhängig von der Eingabe des Benutzers ausführen, ein Datenübertragungsprogramm muß im allgemeinen die Länge der Zeitspanne für das Warten auf eine Antwort messen können. Nach dem Zeitintervall wird ein *Timeout*-Fehler gemeldet. Eine weitere Anwendung ist die Anzeige einer Hilfsmeldung oder die Ausführung einer Standardaktion, falls der Benutzer nicht in einem angemessenen Zeitraum antwortet. OS/2 bietet mehrere Möglichkeiten zur Realisierung von Echtzeitanwendungen. Anwendungen sollten auf diese Leistungsmerkmale zurückgreifen und Polling oder Zeitverzögerungsschleifen vermeiden, da die verstreichende Zeit vom Nutzungsgrad des Systems und der CPU-Geschwindigkeit abhängt. Hinzu kommt, daß dadurch die Ausführung von Threads mit niedrigerer Priorität unterbunden wird.

Zeitintervalle werden in OS/2 in der Größenordnung *Millisekunden* behandelt, um das Konzept eines Zeitintervalles vom physikalischen Mechanismus (periodische Zeitgeberinterrupts), der Zeitintervalle mißt, zu unterscheiden. Ein Zeitintervall kann oft bis zu einer Millisekunde genau angegeben werden. Das bedeutet aber nicht, daß diese Genauigkeit tatsächlich vom System erreicht wird.

In den meisten Systemen verwendet OS/2 Version 1.0 einen periodischen Uhrzeitinterrupt mit einer Frequenz von 32 Hz (32 Ereignisse in der Sekunde). Dies bedeutet, daß OS/2-Zeitintervalle in Einheiten von 31,25 Millisekunden mißt. Folglich kann eine Zeitangabe auch nicht genauer erfolgen. Falls beispielsweise ein Prozeß für 25 Millisekunden unterbrochen werden soll,

erkennt OS/2, daß die Anforderung nach dem letzten Systemtakt erfolgt sein muß. Es kann aber nicht gesagt werden, nach welchem Zeitraum nach dem letzten Systemtakt die Anforderung erfolgte (die Unsicherheit beträgt auf jeden Fall weniger als 31,25 Millisekunden). Nach der Ausführung der Unterbrechungsanforderung tritt ein weiterer Systemtakt auf. Wieder kann OS/2 nicht feststellen, wieviel Zeit seit der Unterbrechungsanforderung und dem neuen Systemtakt vergangen ist. Aus diesem Grunde verwendet OS/2 einen einfachen Algorithmus: Bei jedem Systemtakt dekrementiert das Betriebssystem jeden Timeout-Wert im System um den Systemtaktintervall (üblicherweise 31,25 Millisekunden). Die 25-Millisekunden-Unterbrechungsanforderung kann also nach einer Millisekunde (oder weniger) oder nach spätestens 31,25 Millisekunden zurückkehren. Eine Anforderung zum Blockieren für 33 Millisekunden kann nach 32 Millisekunden oder in einer Zeitspanne von bis zu 62,5 Millisekunden zurückkehren.

Es liegt auf der Hand, daß die OS/2-Zeitgeberfunktionen für menschlich bedeutende Zeitintervalle entwickelt wurden, wobei eine Ungenauigkeit von 1/32 Sekunde kein bedeutsamer Fehler ist. Für genaue Zeitmessungen können sie nicht verwendet werden. Unabhängig von der Auflösung des Zeitgebers verhindert der Zeitteilungsmechanismus des Systems die Implementation eines hochgenauen Zeitmessers für kurze Intervalle. Selbst mit einem Zeitgebersystemaufruf, der einen genauen Intervall messen kann, könnte es vorkommen, daß der aufrufende Thread die Ausführung nicht rechtzeitig wiedererlangt, da ein Thread höherer Priorität zwischenzeitlich ausgeführt wird.

In einigen Systemaufrufen sind Formen der OS/2-Zeitgeberroutinen eingebaut. So unterstützen beispielsweise Aufrufe zum Blockieren einer Semaphore ein Argument, das dem aufrufenden Programm die Spezifikation eines *Timeout*-Wertes erlaubt. Nach dem Verstreichen des spezifizierten Zeitintervalls kehrt der Aufruf mit einem *Timeout*-Fehlercode zurück. Einige Threads verwenden dieses Leistungsmerkmal zum Schutz. Falls bei einem Semaphorenaufruf ein *Timeout*-Fehler auftritt, kann der Thread das Warten beenden und eine Fehlermeldung anzei-

gen oder eine andere Aktion ausführen. Andere Threads können dieses Leistungsmerkmal in der Erwartung eines Timeouts verwenden: Hier dient das *Timeout*-Leistungsmerkmal der Ausführung periodischer Tasks, wobei die Semaphore als Notflagge verwendet wird. Ein anderer Thread im System kann einen Notaufruf an den Zeitgeber-Thread durch das Löschen einer Semaphore auslösen.

Die Blockierung einer Semaphore zur Verzögerung um ein Zeitintervall ist nicht nötig, der *DosSleep*-Aufruf erlaubt einem Thread die Blockierung für eine willkürliche Zeitdauer unter Berücksichtigung des Quantifizierungsfehlers des Zeitgebers. *DosSleep* mißt die Zeitintervalle auf synchrone Weise: Der Thread wird vom Betriebssystem beschäftigt gehalten, bis der Zeitintervall verstrichen ist. Wenn dazu Thread 1 verwendet wird, kann dieser Vorgang durch ein Signal gestört werden. OS/2 stellt auch einen asynchronen Zeitgeberservice zur Verfügung, der die Zeitmessung parallel zur normalen Ausführung des Threads erlaubt. Der *DosTimerAsync*-Aufruf wird mit einem *Timeout*-Intervall (wie *DosSleep*) und mit der Nummer einer Systemsemaphore ausgeführt. Der Aufruf kehrt sofort zurück und nach dem Verstreichen des Zeitintervalls wird die Systemsemaphore freigegeben. Ein Prozeß kann das *Polling* auf eine Semaphore verwenden, um festzustellen, ob die Zeit verstrichen ist und/oder kann eine Semaphore zum selben Zweck blockieren. Falls ein Prozeß mehrere Threads enthält, können einige blockieren, während andere das *Polling* durchführen.

Hinweis: Im Gegensatz zu den meisten Semaphorenanwendungen arbeiten die Zeitgeberfunktionen nur mit Systemsemaphoren. RAM-Semaphoren können nicht verwendet werden, da der Zugriff zu einem Zeitpunkt nicht sofort möglich ist, wenn der Speicherbereich, der die RAM-Semaphore enthält, ausgelagert ist.

Der *DosTimerStart*-Aufruf ist mit dem *DosTimerAsync*-Aufruf bis auf eine Ausnahme identisch. Die Ausnahme: Die Semaphore wird wiederholt nach einem bestimmten Zeitintervall freigegeben, bis ein entsprechender *DosTimerStop*-Aufruf ausgeführt wird. *DosTimerStop* gibt die Semaphore frei. Der Prozeß muß sie wieder setzen, bevor sie erneut freigegeben werden kann.

Keines der oben genannten Leistungsmerkmale mißt die Tageszeit oder ein anderes Zeitintervall exakt. Es wurde schon erwähnt, daß bei der Ausführung eines Treads mit höherer Priorität, der genügend CPU-Zeit für sich beansprucht, unvorhersehbare Verzögerungen auftreten können. Selbst *DosTimerStart* bietet keine Sicherheit vor dem "Verpassen" von Systemtakten. Falls die CPU für eine genügend große Periode nicht verfügbar ist, kann der Prozeß die Semaphore nicht setzen, um die nächste Löschung der Semaphore zu bemerken. Anwendungen, die eine präzise Zeitmessung benötigen, können die Zeitwerte im lokalen Informationssegment (*infoseg*) verwenden. Eine zeitkritische Anwendung sollte einen Thread zur Verwaltung der zeitkritischen Aufgabe besitzen, und zwar auch dann, wenn die Anwendung mit einer niedrigen Priorität im Hintergrund abläuft. Dadurch ist sichergestellt, daß zeitkritische Ereignisse aufgrund von Vordergrund-Threads mit hoher Priorität nicht verpaßt werden. Eine solche Anwendung muß so gestaltet werden, daß ein Zeitgeberereignis mit hoher Priorität nicht selbst signifikante Teile der CPU-Zeit beansprucht. Die Zeitgeberereignisse sollten lediglich protokolliert werden, der Thread mit normaler Priorität erledigt die eigentliche Aufgabe des Prozesses.

Kapitel 17

Einheitentreiber und Hardwarefehler

Da es sich bei OS/2 um ein Multitasking-Betriebssystem handelt, müssen die verwendeten Einheitentreiber wesentlich komplexer sein als die unter MS-DOS eingesetzten. Auch die Behandlung von Geräteeinheitenfehlern (Hardwarefehlern) ist in einer Multitasking-Umgebung schwieriger als in einem Singletasking-System.

17.1 Einheitentreiber

Im folgenden wird ein Überblick über die Einheitentreiber gegeben, wobei insbesondere auf die Architekturelemente eingegangen wird. Die Entwicklung eines Einheitentreibers ist eine komplizierte Aufgabe, da sich rasch Fehler einschleichen können, die für den Programmierer nicht sofort offensichtlich sind. Die in Einheitentreibern auftretenden Fehler zeigen sich im allgemeinen nicht beim Austesten, sondern erst viel später bei der Arbeit mit dem System. Das vorliegende Kapitel soll weder eine vollständige Erläuterung der Einheitentreiber darstellen, noch soll es Richtlinien zur Entwicklung von Einheitentreibern geben. Schlagen Sie gegebenenfalls im OS/2-Referenzhandbuch nach.

Die Rolle der Einheitentreiber in Bezug auf die Geräteunabhängigkeit wurde bereits erläutert. Ein Einheitentreiber ist ein Programm, das Ein-/Ausgabeanforderungen, die in einer standardisierten, geräteunabhängigen Art erfolgen, in die nötigen Operationen für eine vorgegebene Hardwarekomponente (zum Beispiel eine Festplatte) umwandeln. Der Einheitentreiber über-

nimmt auch Daten und Statusinformationen von der Hardware (im hardwarespezifischen Format) und wandelt diese Informationen in eine für das Betriebssystem verständliche Form um.

Die Einheitentreiberarchitektur umfaßt zwei Schlüsselelemente:

- Jede Hardwareeinheit besitzt einen eigenen Treiber zum Verbergen der spezifischen Details der Einheit vor dem Betriebssystem.
- Einheitentreiber sind im Betriebssystem nicht fest verankert. Sie werden beim Start des Systems installiert.

Der zweite Punkt ist besonders beachtenswert. Bei festeingebauten Einheitentreibern wäre die Isolierung des einheitenabhängigen Codes in Paketen eine gute Programmiertechnik, für den Benutzer aber von wenig Interesse. OS/2 würde auf einem einmal konfigurierten System nur mit bestimmten Peripheriegeräten laufen. Da die Einheitentreiber beim Start des Systems dynamisch installiert werden, kann OS/2 mit einer Vielzahl von Einheiten (auch solchen, die bei der Entwicklung von OS/2 noch nicht existierten) betrieben werden, solange ein entsprechender Einheitentreiber für die Einheit installiert wird. Beachten Sie, daß Treiber nur beim Systemstart aufgenommen werden können, eine nachträgliche Installation ist nicht möglich. Der Grund: In zukünftigen sicheren Umgebungen würde die Möglichkeit einer dynamischen Einheitentreiberinstallation für Anwendungsprogramme die Möglichkeit zur Verletzung der Systemsicherheit einräumen.

Die Darstellung von Einheitentreibern als Übertragungssoftware, die zwischen Geräteeinheit und Betriebssystem vermittelt, ist stark vereinfachend. Tatsächlich sind Einheitentreiber für die Isolierung nahezu aller einheitenspezifischen Informationen verantwortlich. Einheitentreiberroutinen bedienen Interrupts, die von dazugehörigen Einheiten erzeugt werden, und sie arbeiten zum Task-Zeitpunkt (das heißt, wenn kein Interrupt abläuft). Der Entwickler eines Einheitentreibers muß festlegen, ob die Einheit gegebenenfalls überwacht werden soll (*monitoring*) und die hierfür erforderliche Unterstützung in den Treiber einbau-

en. Benötigt ein Anwendungsprogramm direkten Zugriff auf die Ein-/Ausgabeports einer Geräteeinheit oder auf den in einer besonderen Form abgelegten Speicherbereich (Bildschirmspeicher der CGA oder EGA), stellt der Treiber die erforderlichen Prozesse dafür bereit. Im Treiber wird auch festgelegt, ob mehrere Prozesse gleichzeitig auf die Einheit zugreifen dürfen; der Treiber erlaubt oder unterbindet dann den Mehrfachzugriff.

17.1.1. Einheitentreiber und OS/2-Kommunikation

Da Einheitentreiber sowohl OS/2 aufrufen als auch von OS/2 aufgerufen werden können, und da Hardwareinterrupts effizient abgearbeitet werden müssen, laufen sie in Ring 0 ab. Dies bedeutet, daß Einheitentreiber fehlerlos sein müssen, da sie nahezu beliebig viele Daten und Anwendungen in einem System zerstören können.

Der Aufruf eines Einheitentreibers für OS/2 ist einfach. Da das Betriebssystem den Einheitentreiber in den Speicher lädt, kennt es die Adresse des Einsprungpunktes und kann den Einheitentreiber direkt aufrufen. Der Aufruf von OS/2 durch einen Einheitentreiber ist schwieriger, da der Treiber die von OS/2 belegten Speicherstellen nicht kennt und auch keine Kontrolle über die Speicherdeskriptortabellen (LDT und GDT) besitzt. Bei der Initialisierung des Einheitentreibers liefert das Betriebssystem daher dem Treiber die Adresse des *DevHlp*-Einsprungpunktes von OS/2. Einheitentreiber rufen diese Adresse für den Zugriff auf eine Vielzahl von OS/2-Routinen auf, die *DevHlp*-Routinen genannt werden. Die OS/2-*DevHlp*-Adresse ist eine Referenz auf einen GDT-Selektor, so daß die Adresse zu jedem Zeitpunkt gültig ist (im *Protect*-Modus, im *Real*-Modus, bei einem Interrupt und bei der Initialisierung eines Einheitentreibers). Einige *DevHlp*-Funktionen können nur in bestimmten Modi verwendet werden, das Leistungsmerkmal des *DevHlp* ist aber, daß es grundsätzlich immer verfügbar ist.

Hinweis: Ein GDT-Selektor kann (streng betrachtet) im Real-Modus nicht gültig sein, da die GDT hier nicht verwendet wird. OS/2 benutzt die Technik des Tiling, so daß der Selektor (bei der Verwendung als Segmentadresse im Real-Modus) den physikalischen Speicherbereich anspricht, der dem Protect-Modus-Segment entspricht.

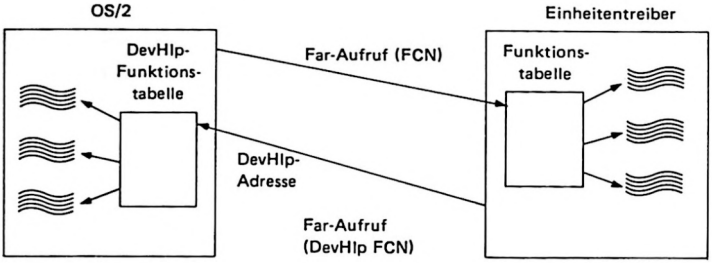


Bild 17.1 Aufrufverknüpfungen des Einheitentreibers

Warum verwenden Einheitentreiber nicht einfach dynamische Verknüpfungen zum Zugriff auf OS/2-Dienste (wie Anwendungssoftware)? Die OS/2-Kernel-Dynlink-Schnittstelle wurde für Prozesse entworfen, die im Benutzermodus (Ring 3) ablaufen und den Kernel (Ring 0) aufrufen. Mit anderen Worten: Sie wurde für "Außenstehende" zum Aufruf von Kernelroutinen entworfen. Einheitentreiber sind aber immer im Kernel. Sie laufen in Ring 0, dem Kernel-Modus, und während des Interrupts ab. Die Situation kann so strukturiert werden, daß Einheitentreiber Dynlink-Aufrufe ausführen und der Programmteil beim OS/2-Einsprungpunkt die Einheitentreiberanforderung erkennt und entsprechende Aktionen vornimmt. Durch diesen "Extraprogrammteil" würden aber alle anderen Systemaufrufe der normalen Anwendung verlangsamt, gleiches gilt für Serviceaufrufe von Einheitentreibern. Aus diesem Grund besitzen Einheitentreiber eine "Hintertür", die den effizienten Einsprung in OS/2 erlaubt. In Bild 17.1 sind die Aufrufverknüpfungen zwischen OS/2 und einem Einheitentreiber gezeigt. OS/2 ruft nur einen Einsprungpunkt im Einheitentreiber auf und stellt einen Funktionscode bereit, den der Einheitentreiber zur Adressierung einer Verteilertabelle verwendet. Die Adresse des Einsprungpunktes wird OS/2 beim Laden des Einheitentreibers bekannt. Der Einheitentreiber wiederum ruft nur eine OS/2-Adresse auf, den *DevHlp*-Einsprungpunkt. Auch hier wird ein Funktionscode bereitgestellt, der zur Adressierung einer Verteilertabelle verwendet wird. Dem Einheitentreiber wird die Adresse beim Ent-

gegennehmen seines Initialisierungsaufrufes von OS/2 übergeben. Nicht gezeigt ist der Interrupteinsprungpunkt des Einheitentreibers.

17.1.2 Modell zur Programmierung eines Einheitentreibers

Das Programmiermodell für Einheitentreiber unter MS-DOS gestaltet sich einfach. Einheitentreiber werden zur Ausführung einer Funktion aufgerufen und geben die Kontrolle zurück, nachdem die Funktion vollständig ausgeführt ist oder ein nicht behebbarer Fehler aufgetritt. Bei interruptgesteuerten Einheiten befände sich die CPU in einer Schleife im Einheitentreiber und würde auf den Einsprung in die interruptbearbeitende Routine des Treibers warten. Nach der Beendigung der Operation würde die interruptbearbeitende Routine eine eigene Flagge in der Warteschleife zur Unterbrechung der CPU setzen.

Das OS/2-Einheitentreibermodell ist wesentlich komplizierter, da OS/2 ein Multitasking-System ist. Selbst wenn der Thread, der den Aufruf an den Einheitentreiber auslöst, keine weiteren Aufgaben außer dem Warten auf die Rückkehr vom Systemaufruf hat, können andere Threads im System diese Zeit nutzen. Hinzu kommt, daß zwei oder mehr Threads den Einheitentreiber gleichzeitig aufrufen können. Um diesen letzten Punkt zu verstehen, muß man Kenntnisse über die interne Ausführung von OS/2 haben.

Vom konzeptionellen Ansatz her gesehen funktioniert OS/2 eher wie eine Unterroutinenbibliothek, denn wie ein Prozeß. Die einzigen verteilbaren Elemente im System sind Threads, und alle Threads gehören Prozessen an. Wenn ein Thread eines Prozesses OS/2 aufruft, führt dieser Thread OS/2-Code aus. Es gibt einige Ausnahmen, die an dieser Stelle aber nicht wichtig sind. Der Vorgang läßt sich am Beispiel eines Schnellrestaurants plastisch erläutern. Man geht zum Schalter und gibt seine Bestellung auf. Anschließend zieht man sich selbst einen Kittel an, geht um die Theke herum und bereitet die eigene Bestellung zu. Nachdem das Essen fertiggestellt ist, wird der Kittel ausgezogen, auf die andere Seite der Theke gegangen und das fertige

Menü mitgenommen. Die Theke stellt hier die Grenze zwischen Ring 0 (Kernel-Modus) und Ring 3 (Anwendungsmodus) dar, der Kittel entspricht dem Privileg, das zum Arbeiten hinter der Theke nötig ist.

Selbstverständlich ist OS/2 eintrittsvariant (*reentrant*), das heißt, zu einem Zeitpunkt können mehrere Threads verschiedene Aufgaben unter OS/2 ausführen. Jeder davon verrichtet aber nur die Arbeit für einen Prozeß - für den Prozeß, zu dem der Thread gehört. Zurück zu unserem Vergleich mit dem Schnellrestaurant: Hinter der Theke befinden sich viele Angestellte (in Kitteln), von denen aber jeder nur für seine eigene Bestellung arbeitet. Dies vereinfacht die internen Organisationsvorgänge von OS/2: Jede Instanz eines Programmteils erledigt nur die Aufgabe für einen Klienten. Falls ein Teil des Programms auf ein Ereignis warten muß, wird eine Blockierung (zum Beispiel das Warten auf eine Semaphore) ausgeführt und anschließend (hier nach der Rückkehr der Semaphore) die Bearbeitung wieder aufgenommen. Threads im Kernel, die auf Ressourcen zugreifen wollen, tun dies über interne Semaphoren. Der Zugriff auf die Semaphoren erfolgt auf der Basis der Priorität, ähnlich der Ausführung im Anwendungsmodus.

OS/2 macht quasi keine Unterscheidung zwischen einem Thread, der im Kernel und einem Thread, der außerhalb des Kernels abläuft: Die LDT des Prozesses bleibt weiterhin gültig, der Thread kann (solange er im Kernel ist) auf jede Speicherposition zugreifen, die dem Prozeß auch im Anwendungsmodus verfügbar ist. Hinzu kommt, daß der Zugriff auf Speicher möglich ist, der nur Ring-0-Programmen zugänglich ist. Der einzige Unterschied, den der Scheduler zwischen Threads innerhalb und außerhalb des Kernels macht, besteht darin, daß ein Thread im Kernel niemals unterbrochen wird. Dadurch werden die im Kernel-Modus benötigten Schutzvorkehrungen für kritische Bereiche vereinfacht: Bei der Ausführung des Kernel-Codes schaltet der Scheduler nur um, wenn sich die CPU selbst blockiert. Solange sich das Kernel-Programm nicht selbst blockiert (auf eine Semaphore wartet oder eine Unterroutine aufruft, die auf eine Semaphore wartet), braucht nicht befürchtet zu werden,

daß ein anderer Thread auf einen kritischen Bereich Zugriff erhält.

Hinweis: Obwohl Hardwareinterrupts auftreten können, bleiben die kritischen Bereiche geschützt.

Ruft OS/2 einen Einheitentreiber zum Task-Zeitpunkt auf, erfolgt dies mit dem Thread, der den OS/2-Kernel-Code ausführt. Das ist der Thread, der zum Klientenprozeß gehört und den Dienstleistungsaufruf initiiert hat. Der Task-Teil des Einheitentreibers läuft in Ring 0 in der Umgebung des Klientenprozesses ab. Die LDT des Klienten ist (wie auch alle Adressen des Klientenprozesses) aktiv, der Einheitentreiber kann von einem anderen Task-Thread nicht vorzeitig unterbrochen werden (wohl aber durch einen Interrupt-Service), solange keine Blokkierung über eine *DevHlp*-Funktion oder der Rücksprung zu OS/2 erfolgt.

OS/2-Einheitentreiber werden in zwei Kategorien unterteilt: es gibt zeichenorientierte und blockorientierte Geräteeinheiten. Diese Terminologie stammt noch aus Zeiten der Entwicklung von UNIX, darf aber unter OS/2 nicht wörtlich genommen werden, da auch zeichenorientierte Operationen bei blockorientierten Einheiten ausgeführt werden können. Die eigentliche Unterscheidung besteht darin, daß zeichenorientierte Einheitentreiber ihre Ein- und Ausgabe synchron ausführen; das heißt, die Operationen werden in der Reihenfolge des Prinzips FIFO (*first in first out*) ausgeführt. Blockmoduseinheitentreiber können asynchron sein, das heißt, sie können Ein-/Ausgabeanforderungen in einer anderen Reihenfolge ausführen, als diese beim Einheitentreiber ankommen. Ein herkömmliches, serielles und zeichenorientiertes Gerät wie ein Drucker darf die Reihenfolge der Anforderungen nicht verändern, da dies die Ausgabe unbrauchbar machen würde. Eine blockorientierte Einheit (wie ein Diskettenlaufwerk) kann die Reihenfolge von zwei Sektorenlesevorgängen problemlos verändern.

In Bild 17.2 sehen Sie einen Algorithmus für zeichenorientierte Einheitentreiber. Weitere Details finden Sie im Referenzhandbuch für Einheitentreiber. OS/2 ruft die Einheitentreiber mit

einer Anforderung auf, wie sie am oberen Rand des Bildes zu sehen ist. Ist der Einheitentreiber mit einer anderen Anforderung beschäftigt, sollte der anfordernde Thread die Einheit über eine RAM-Semaphore bis zur Verfügbarkeit blockieren. Wird die Einheit frei, führt der Thread die geforderte Operation aus. Teilweise kann die Aufgabe zum Task-Zeitpunkt erfüllt werden (wie ein IOCTL-Aufruf, der die Anzahl der Zeichen im Eingabepuffer bestimmt), häufiger initiiert der Task-Thread jedoch eine Operation und die Aufgabe wird zum Interrupt-Zeitpunkt beendet. Falls der Task-Thread auf einen Interrupt-Service warten muß, sollte er eine RAM-Semaphore (im Datenbereich des Einheitentreibers) blockieren, die vom Programmteil zum Interrupt-Zeitpunkt freigegeben wird. Sobald der letzte angeschlossene Einheiten-Interrupt ausgeführt wird und die Operation vervollständigt ist, gibt der Interrupt-Code die RAM-Semaphore frei. Der Task-Thread wird weiter abgearbeitet und kehrt mit den entsprechend gesetzten Statusbits im Anforderungsblock zu OS/2 zurück.

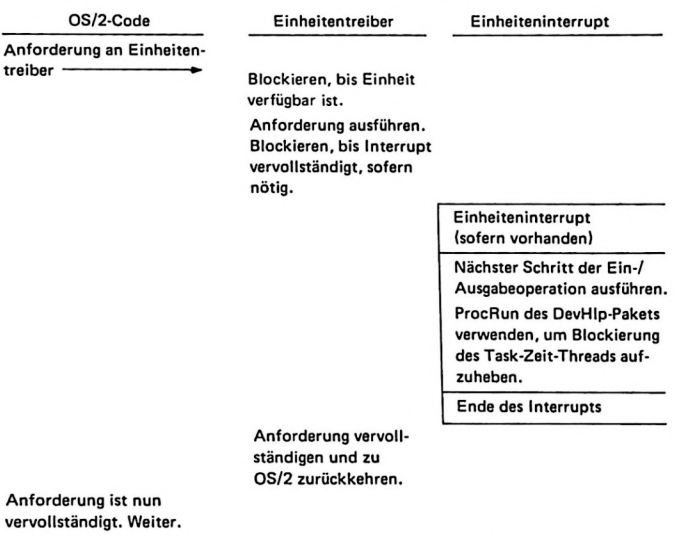


Bild 17.2 Vereinfachtes Einheitentreibermodell für zeichenorientierte Einheiten

In Bild 17.3 ist der Algorithmus für blockorientierte Einheiten dargestellt. Der Aufbau gleicht dem für zeichenorientierte Einheiten, ist aber aufgrund der asynchronen Bearbeitung bei Einheiten mit wahlfreiem Zugriff komplizierter. Da Anforderungen in einer beliebigen Reihenfolge bearbeitet werden können, pflegen die meisten blockorientierten Einheitentreiber eine interne Arbeitswarteschlange, der neue Aufgaben hinzugefügt werden. Üblicherweise wird eine *DevHlp*-Funktion verwendet, um den Inhalt der Warteschlange nach Sektornummern zu sortieren, so daß die Bewegung des Schreib-/Lesekopfes minimiert wird.

Am einfachsten kann man dies verstehen, wenn man sich einen blockorientierten Einheitentreiber vorstellt, der aus N Threads (ein Interrupt-Thread, der die eigentliche Arbeit ausführt, und $N-1$ Task-Threads, die die Aufgaben in der Warteschlange ausführen) besteht. Sobald eine Anforderung durch einen Task-

Thread beim Treiber ankommt, wird die Anforderung in die Schlange aufgenommen und an OS/2 weitergegeben. Zu einem späteren Zeitpunkt ruft die interruptbearbeitende Routine des Einheitentreibers die *DevHlp*-Funktion *DevDone* auf, um OS/2 von der Vervollständigung einer Operation zu informieren. Die Rückkehr zu OS/2 vor der Vervollständigung der Operation ist möglich, da der Anforderungsblock Statusbits enthält, die den Stand der Operation anzeigen.

Teilweise muß OS/2 auf die Beendigung einer Operation (zum Beispiel beim Lesen eines Verzeichnisses) warten. Kehrt der Einheitentreiber mit einer nicht vollständig bearbeiteten Operation zurück, wartet OS/2, bis diese beendet ist. Unter anderen Umständen (etwa beim Löschen eines Zwischenpuffers) muß OS/2 nicht auf die Vervollständigung der Operation warten. Andere Aufgaben oder sogar eine neue Anforderung an denselben Treiber können ausgeführt werden, wobei allerdings ein neuer Anforderungsblock verwendet werden muß (da der alte Anforderungsblock noch in Benutzung ist). Dieses Konzept erlaubt dem System eine optimierte Parallelverarbeitung und steigert den Systemdurchsatz.

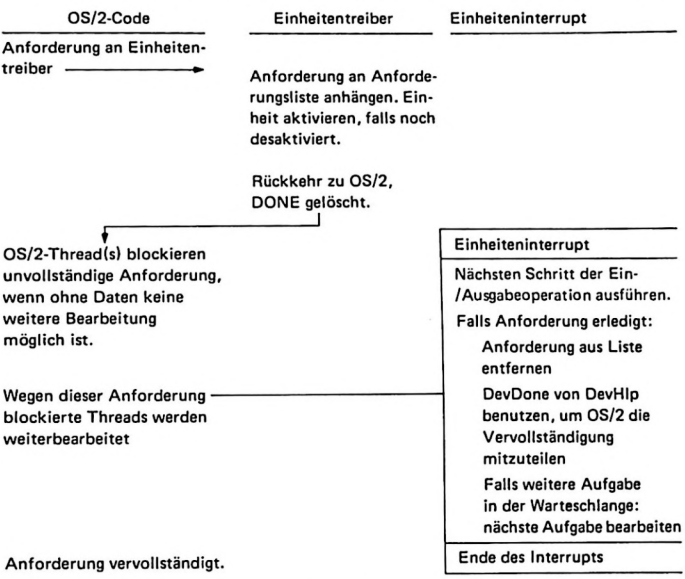


Bild 17.3 Modell eines blockorientierten Einheitentreibers

Ein blockorientierter Einheitentreiber besteht aus *N-1* Task-Threads und einem Interrupt-Thread. Der Ausdruck Interrupt-Thread ist hier irreführend, da dies kein tatsächlicher Thread ist (der vom Scheduler verwaltet wird), sondern ein *Pseudo-Thread*, der durch den Hardwareinterruptmechanismus erzeugt wurde.

Ein Beispiel: Für einen Disketteneinheitentreiber liegen vier Anforderungen in einer Schlange vor, die Leseoperation der ersten Anforderung wird gerade bearbeitet. Nach der Beendigung wird die interruptbearbeitende Routine des Einheitentreibers durch einen Hardwareinterrupt (vom Laufwerk-Controller) an-

gesprochen. Dieser Interrupt-Thread, der die interruptbearbeitende Routine des Einheitentreibers ausführt, prüft den Status ab (ob alles in Ordnung ist). Außerdem werden verschiedene *DevHlp*-Routinen aufgerufen, um Mitteilungen über die Vervollständigung einer Anforderung zu übergeben und vollständig bearbeitete Anforderungen aus der Schlange zu löschen. Anschließend wird festgestellt, daß weitere Anforderungen in der Warteschlange anstehen und die Ausführung der nächsten Anforderung begonnen (dies soll hier eine Suchoperation sein). Der Interruptprogrammcode des Einheitentreibers gibt ein Suchkommando an die Hardware weiter und kehrt von diesem Interrupt zurück. Nachdem der Suchvorgang auf der Diskette beendet ist, wird ein weiterer Interrupt erzeugt. Der Programmcode zur Bearbeitung des Interrupts erkennt, daß die Suche erfolgreich war, gibt eine Lese- oder Schreiboperation an den Controller weiter und beendet den Vorgang.

Die wiederholte Aktivierung der interruptbearbeitenden Routine eines Einheitentreibers ähnelt einem Thread, unterscheidet sich aber von diesem in zwei wesentlichen Punkten.

- Bei jedem Einsprung in die interruptbearbeitende Routine wird dieser ein neuer Stapel zugewiesen. Für einen Task-Thread werden die Registerinhalte und der Stapel vom System gesichert; keines von beiden wird zwischen zwei Interrupts für eine interruptbearbeitende Routine gespeichert. Ein Task-Thread verfolgt die Inhalte der Adresse CS:IP, der Registerinhalte und der Stapelinhalte bei der Verarbeitung. Eine interruptbearbeitende Routine hingegen muß Informationen bezüglich des aktuellen Status über statische Werte im Datensegment des Einheitentreibers erlangen. Interruptbearbeitende Routinen implementieren üblicherweise ein Statussystem und besitzen Informationen zum aktuellen Status im Datensegment des Einheitentreibers.
- Ein echter Thread bleibt existent, bis er explizit beendet wird. Ein Interrupt-Thread stellt hingegen lediglich "die Vortäuschung" eines Threads dar, die durch wiederholte Interrupts aufrecht erhalten wird. Falls die Ausführung ei-

ner interruptbearbeitenden Routine der Laufwerk-Hardware keinen Befehl zur Generierung eines weiteren Interrupts gibt, besteht der Pseudo-Interrupt-Thread nicht mehr, nachdem die interruptbearbeitende Routine von ihrer Aufgabe zurückkehrt.

In der Algorithmusbeschreibung blockorientierter Treiber wurde ein Detail ausgelassen. Falls bei der Ankunft einer neuen Anforderung kein Laufwerkszugriff erfolgt, wird die Anforderung in die Warteschlange eingereiht. Es gibt allerdings keinen Pseudo-Thread zum Interrupt-Zeitpunkt, der die Anforderung bearbeiten könnte. Aus diesem Grund müssen sowohl Task- als auch Interrupt-Teile eines Einheitentreibers zur Initiierung einer Operation fähig sein. Empfohlen wird die Verwendung eines Softwarestatussystems zur Steuerung der Hardware. Dabei muß sichergestellt werden, daß das Statussystem (oder zumindest eines Teiles davon - die Startoperationen) sowohl zum Task- als auch zum Interrupt-Zeitpunkt aufgerufen werden kann. Der Algorithmus wird modifiziert, so daß nach der Einreihung einer Anforderung durch den Task-Teil eines blockorientierten Einheitentreibers sichergestellt wird, daß die Einheit (oder das Statussystem oder der Interrupt-Pseudo-Thread) aktiv ist. Falls die Einheit gerade nicht angesprochen wird, löst der Task-Thread im Einheitentreiber die Operation durch das Setzen des Anfangsstatus des Statussystems aus und kehrt anschließend zu OS/2 zurück. Dadurch wird der Vorgang wieder "in Schwung gebracht" und der Pseudo-Interrupt-Thread führt die Ausführung der Anforderungen in der Schlange fort, bis diese leer ist.

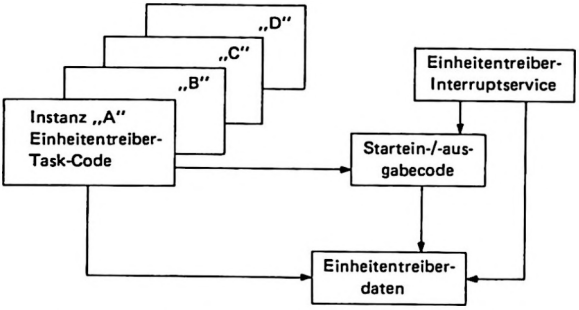


Bild 17.4 Programmstruktur eines Einheitentreibers

In Bild 17.4 sehen Sie einen Überblick über die Einheitentreiberarchitektur von OS/2. Jeder Einheitentreiber besteht aus einem Task-Teil, einem Interrupt-Teil (falls die Einheit Interrupts erzeugt) und dem Startcode für Operationen, der in beiden Modi ausgeführt werden kann. Der Datenbereich des Treibers enthält üblicherweise Statusinformationen, Flaggen und Semaphoren zur Bearbeitung der Kommunikation zwischen Task-Teil und Interrupt. In Bild 17.4 wird auch gezeigt, daß der Task-Teil eines Einheitentreibers mehrere Instanzen besitzen kann. Er kann von mehreren Threads gleichzeitig aufgerufen werden (ähnlich einer gemeinsam verwendeten Dynlink-Bibliotheksroutine). Im Gegensatz zu einer Dynlink-Bibliothek besitzt ein Einheitentreiber aber kein Instanzdatensegment. Das Datensegment des Einheitentreibers ist global, wobei alle Ausführungsinstanzen der Task-Komponente darauf zugreifen können. Wie ein Dynlink-Paket Semaphoren zum Schutz kritischer Bereiche in einem globalen Datensegment einsetzt, so verwendet ein Einheitentreiber Semaphoren zum Schutz von kritischen Datenwerten in dessen Datensegment. Der Einheitentreiber besitzt einen weiteren Thread - den Interruptservice-Thread. Ein Einheitentreiber kann kritische Bereiche während des Zugriffs zum Interrupt-Zeitpunkt durch die Verwendung einer Semaphore nicht schützen, da ein Interruptservice-Thread keine Blockie-

rung ausführen kann. Der Interruptservice muß vervollständigt und der Vorgang beendet werden. Bei der Entwicklung von Einheitentreibern ist die Verwendung kritischer Bereiche durch den Interruptservice-Thread zu vermeiden. Solche Bereiche müssen mit der Befehlsfolge CLI/STI geschützt werden.

17.1.3 Einheitenverwaltung

Einheitentreiber erledigen mehr als nur die Kommunikation mit der Einheit, sie verwalten die Einheit für das System. Der Einheitentreiber wird aufgerufen, sobald ein Prozeß eine Einheit öffnet oder schließt. Einheitentreiber bestimmen, ob eine Einheit von einem oder mehreren Prozessen gleichzeitig verwendet werden kann. Gleichermaßen empfangen Einheitentreiber Anforderungen von Einheitenüberwachungsprozessen über die IOCTL-Schnittstelle und rufen OS/2 über die *DevHlp*-Schnittstelle zur Ausführung der Überwachungsaufgabe auf (sofern notwendig). Einheitentreiber können Prozessen den Zugriff auf Ein-/Ausgabeports, auf Speicherbereiche und/oder auf Kontrollbereiche gewähren; Prozesse fordern diese Leistungsmerkmale über die IOCTL-Schnittstelle an. Der Einheitentreiber bearbeitet die Anforderung über einen *DevHlp*-Dialog mit OS/2. Es gibt einige degenerierte Einheitentreiber, die keine eigentlichen Daten übertragen, sondern nur für die anderen Aufgaben bereitstehen. Der Bildschirmeinheitentreiber ist ein Beispiel hierfür. Bildschirmdaten werden immer direkt in den Anzeigepuffer des VIO-Pakets, der Anwendung oder des Presentation Manager geschrieben. Der Einheitentreiber für den Bildschirm existiert zur Vergabe des direkten Zugriffs, zur Verwaltung von Bildschirmgruppen und ähnlichen Aufgaben.

17.1.4 Dual-Modus

Das letzte Schlüsselelement von Einheitentreibern besteht darin, daß sie im *Dual-Modus* entwickelt sind: Der Treibercode muß zum Task-Zeitpunkt wie zum Interrupt-Zeitpunkt zur Ausführung sowohl im *Protect*- als auch im *Real*-Modus geeignet sein. Der Prozeß der Modusumschaltung zwischen Protect- und Real-Modus ist relativ langsam und dauert länger als 800 Mikrose-

kunden. Falls alle Einheitentreiber und Serviceinterrupts nur im Protect-Modus ablaufen, würde eine normale Diskettenanforderung aus einem Real-Modusprogramm heraus sechs oder mehr Modusumschaltungen erfordern (eine für die Anforderung und fünf für die Interrupts). Folglich müssen alle Einheitentreiber in dem Modus ausführbar sein, in dem die CPU gerade arbeitet.

Solange der Einheitentreiber nicht seinen eigenen Segmentselektor beeinflußt, kann dieser in beiden Modi ausgeführt werden. Das Problem ist hierbei, daß OS/2 bei jedem Aufruf und/oder Interrupt zwischen den Modi umschalten kann und sich die Code- und Datenadressen in den Modi unterscheiden. Ein Einheitentreiber könnte im Protect-Modus mit der Adresse eines Klientenprozesses aufgerufen werden. Beim Auftreten des *Daten-bereit*-Interrupts könnte die CPU im Real-Modus ablaufen und die Adresse des Klientenprozesses wäre nicht länger gültig. Zum einen hat der Segmentselektorteil einer Speicheradresse im Real-Modus eine andere Bedeutung als dies im Protect-Modus der Fall ist. Zum zweiten ist der Selektor des Klientenprozesses in der LDT, die zum Interrupt-Zeitpunkt ungültig ist (weitere Informationen finden Sie im Referenzhandbuch). OS/2 hilft Einheitentreibern bei der Adressierung in einer Umgebung, die zwei Modi annehmen kann, wie folgt:

- Einige Adressen sind in beiden Modi identisch. Der *DevHlp*-Einsprungpunkt, die Adresse des globalen Infosegmentes (*infoseg*) und jede von der *DevHlp*-Funktion *GetDosVar* gelieferte Adresse sind zu jedem Zeitpunkt und in beiden Modi gültig.

- Obwohl der Segmentselektorwert für den Einheitentreibercode und die Datensegmente in jedem Modus unterschiedlich ist, lädt OS/2 die korrekten Werte in die Register CS und DS, bevor der Task- oder Interrupt-Einsprungpunkt des Einheitentreibers angesprochen wird. Solange ein Einheitentreiber diese Werte nicht wiederverwendet, wird er nicht feststellen, daß sie bei unterschiedlichen Aufrufen (gegebenenfalls) unterschiedlich sind.

- OS/2 stellt eine Vielzahl von *DevHlp*-Funktionen bereit, die einem Einheitentreiber die Konvertierung einer Adresse in der Form *Selektor:Offset* in eine physikalische Adresse und umgekehrt ermöglichen. Einheitentreiber können dadurch auch Adressen, die nicht in ihren eigenen Segmenten vorliegen, in physikalische Adressen umwandeln und zum Task- oder Interrupt-Zeitpunkt den Treiber aufrufen. Dabei wird die physikalische Adresse in die dem aktuellen Modus entsprechende Form konvertiert. So werden Probleme bei der Speicherung eines Selektor:Offset-Paares im Protect-Modus verhindert, wenn dieses später in der Form Segment:Offset im Real-Modus verwendet werden soll.

17.2 Hardwarefehler

Es gibt Fehler, die das System weder ignorieren noch korrigieren kann. Ein klassisches Beispiel ist eine offene Laufwerksklappe, die nur vom Benutzer geschlossen werden kann. Das System kann keinen Zugriff auf die Diskette ausführen, solange die Laufwerksklappe nicht geschlossen wird. Ein solcher Fehler wird *Hardwarefehler* genannt.

Die Art und Weise, in der MS-DOS Hardwarefehler behandelt, ist einfach. Im Beispiel der Laufwerksklappe erkennt MS-DOS das Problem, während es mit dem Diskettentreiber kommuniziert. Der Einheitentreiber berichtet das Problem und MS-DOS zeigt einen Text auf dem Bildschirm an - die bekannte Meldung *Abbrechen, Wiederholen, Ignorieren?*. Üblicherweise behebt der Benutzer das Problem und antwortet, MS-DOS fährt mit der Arbeit fort. Häufig wird es aber von Anwendungen nicht gewünscht, daß das System Hardwarefehler automatisch bearbeitet. Dabei kann leicht die Datenintegrität zerstört werden, oder das System will von Problemen beim Schreiben auf Diskette benachrichtigt werden. Eine weitere Möglichkeit ist die fehlerhafte Spezifikation von *Ignorieren* durch den Benutzer. Ebenfalls denkbar ist, daß MS-DOS die Bildschirmdarstellung nicht ohne Kenntnis des Anwendungsprogramms zerstören soll. Um allen diesen Möglichkeiten gerecht zu werden, erlaubt MS-DOS die

Speicherung einer hardwarefehlerbearbeitenden Routine, auf die der Vektor des INT 24 zeigt. Falls die Routine verfügbar ist, ruft MS-DOS diese statt der eigenen hardwarefehlerbearbeitenden Routine auf.

Hinweis: Die Antwort "Ignorieren" wird auf Hardwarefehler häufig gegeben. Mit dem System vertraute Benutzer verstehen die Konsequenzen einer solchen Anwort, die meisten Benutzer setzen "Ignorieren" aber mit dem Lösen des Problems gleich.

Die Anwendung ruft MS-DOS für die Bearbeitung eines Hardwarefehlers ursprünglich über den Vektor INT 21 auf. Das Betriebssystem führt anschließend mehrere interne Aufrufe aus, wobei eine MS-DOS-Routine die hardwarefehlerbehandelnde Routine der Anweisung aufruft. Da MS-DOS nicht generell eintrittsvariant (wiederverwendbar) ist, kann das Anwendungsprogramm den INT 21 von MS-DOS an diesem Punkt nicht nochmals aufrufen; dies würde bedeuten, daß MS-DOS zum gleichen Zeitpunkt zweimal aufgerufen wird. Die Anwendung muß aber vermutlich die Bildschirm- und Tastaturein-/-ausgabe zur Behandlung des Hardwarefehlers ausführen, weshalb MS-DOS teilweise eintrittsvariant gestaltet wurde. Der ursprüngliche Aufruf beschäftigte sich mit der Diskettenein-/-ausgabe, so daß MS-DOS mit einer Bildschirm- oder Tastaturein-/-ausgabe ohne Probleme nochmals aufgerufen werden kann.

Unter OS/2 müssen die im Zusammenhang mit Hardwarefehlern auftretenden Probleme grundsätzlich anders gelöst werden. Im Gegensatz zu Singletasking-Betriebssystemen wie MS-DOS kann OS/2 zum einen die Operation während eines Betriebssystemaufrufs an eine Anwendung nicht unterbinden - dies könnte ein Aufruf sein, der erst nach langer Zeit beendet wird. Zum zweiten treten technische und Sicherheitsprobleme beim Aufruf von Ring 0 (Kernel-Modus) nach Ring 3 (Anwendungsmodus) auf. In einer MS-DOS-Umgebung ist die Bestimmung des den Fehler verursachenden Prozesses einfach, hier läuft nur eine Anwendung ab. Unter OS/2 kann es bei der Bestimmung des fehlerhaften Programms zu Problemen kommen, da mehr als ein Prozeß einen FAT-Sektor oder ein Diskettenverzeichnis modifizieren können. Die erweiterten Puffertechniken von OS/2 können zu einem Zeitpunkt zu einem Hardwarefehler führen, zu

dem kein Prozeß eine Ein- oder Ausgabe ausführt. Nach Lösung all dieser Probleme könnte die Anwendung noch in einer Hintergrundbildschirmgruppe ablaufen und daher keine Meldung auf dem Bildschirm anzeigen; die Tastaturbenutzung ist dabei gleichermaßen nicht möglich. Selbst wenn die Anwendung in einer Vordergrundbildschirmgruppe abläuft, können Bildschirm und Tastatur nur verwendet werden, wenn der den Fehler verursachende Prozeß die Zugriffsrechte darauf besitzt.

17.2.1 Der Hardwarefehler-Daemon

OS/2 unterstützt mehrere Bildschirmgruppen. Der Bildschirmgruppenmechanismus verwaltet die Benutzung von Bildschirm und Tastatur, indem die aktuellen Benutzer - einer für jede Bildschirmgruppe - voneinander isoliert gehalten werden. Offensichtlich ist, daß Bildschirmgruppen zur Dialogausführung aufgrund von Hardwarefehlern verwendet werden müssen, ohne daß dabei die aktuelle Vordergrundanwendung gestört wird. Nur so werden Fehlermeldungen beim Auftreten eines Hardwarefehlers nicht in die Bildschirmdarstellung einer anderen Anwendung geschrieben.

OS/2 besitzt einen permanent aktiven Prozeß mit dem Namen *Hardwarefehler-Daemon*. Beim Auftreten eines Hardwarefehlers versucht OS/2 nicht herauszufinden, welcher Prozeß den Hardwarefehler verursacht hat; statt dessen wird der Hardwarefehler-Daemon aufgerufen. Dieser führt eine besondere Form der Bildschirmgruppenumschaltung zu einer reservierten Bildschirmgruppe für Hardwarefehler aus, zeigt Meldungen an und liest die Benutzereingabe. Da dies der einzige Prozeß dieser Bildschirmgruppe ist, führen Bildschirm- und Tastaturbenutzung nicht zu Zugriffskonflikten. Der vorher laufende Vordergrundprozeß wird durch den Bildschirmgruppenmechanismus temporär in den Hintergrund geschaltet.

Zwischenzeitlich ist der den Hardwarefehler verursachende Thread im Kernel blockiert und wartet darauf, daß der Hardwarefehler-Daemon eine Antwort vom Benutzer übernimmt. Der Thread, der den Hardwarefehler behandelt, ist niemals der

Thread, aufgrund dessen der Hardwarefehler entstand. Der Kernel ist für andere Threads bereits ohne Einschränkungen wiederverwendbar. Der Thread des Hardwarefehler-Daemon kann OS/2 beliebig aufrufen. Nachdem der Benutzer das Problem korrigiert hat und dem Hardwarefehler-Daemon antwortet, sendet dieser die Anwort zum Kernel, der dem den Hardwarefehler entdeckenden Thread die Ausführung der spezifizierten Aktion erlaubt. Dieser Thread versucht die Operation entweder nochmals oder erzeugt einen Fehlercode und der Hardwarefehler-Daemon versetzt das System wieder in die Originalbildschirmgruppe. Der Bildschirmgruppenprogrammcode führt den üblichen Trick zum Laden der Bildschirmdarstellung mit dem vorherigen Status aus. In Bild 17.5 wird die Behandlung eines Hardwarefehlers gezeigt. Ein Thread stellt einen Hardwarefehler fest, während er sich im OS/2-Kernel befindet. Der Thread blockiert zu diesem Zeitpunkt, während der zuvor beschäftigt gehaltene Thread des Hardwarefehler-Daemons freigegeben wird. Der Hardwarefehler-Daemon führt die modifizierte Bildschirmumschaltung bei (1) aus, zeigt eine Meldung an, übernimmt die Antwort des Benutzers, lädt die Bildschirmgruppe der Anwendung (2) und springt wieder in den OS/2-Kernel zurück. Der Antwortcode wird an den blockierenden Thread übergeben, der seine Arbeit wieder aufnimmt.

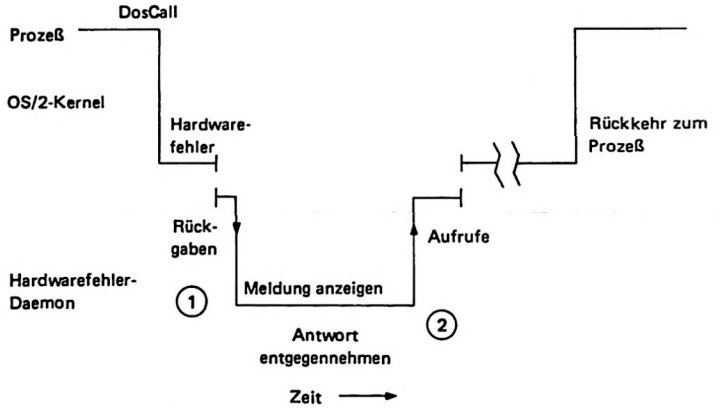


Bild 17.5 Bearbeitung von Hardwarefehlern

Obwohl der am häufigsten vorkommende Grund für Hardwarefehler ein Laufwerksproblem ist - beispielsweise eine geöffnete Laufwerksklappe oder ein Datenträgerfehler - wird auch bei anderen Ereignissen, die die Einbeziehung des Benutzers erfordern, ein Hardwarefehlermechanismus benutzt. So setzt beispielsweise das Datenträgerverwaltungspaket den Hardwarefehlermechanismus zur Anzeige seiner Meldungen ein. Wie bereits erwähnt, können MS-DOS-Anwendungen, die in einer Kompatibilitätsbox ablaufen, Probleme (gesperrte Dateien) entdecken, die sie nicht verstehen. Als Folge wird die Anwendung aber nicht aufgrund mysteriöser Fehler beendet, vielmehr verwendet OS/2 den Hardwarefehlermechanismus zur Information des Benutzers. Anwendungsprogramme in der Kompatibilitätsbox sehen das (nicht vorhandene) Betriebssystem MS-DOS, das in Wirklichkeit OS/2 ist. OS/2 kann den INT 24-Hardwarefehlermechanismus - unverändert - nicht unterstützen, auch nicht für MS-DOS-Anwendungen, die im Real-Modus ablaufen. Der Hardwarefehler-Daemon behandelt auch solche Hardwarefehler.

17.2.2 Hardwarefehlerbehandlung durch Anwendungen

In einigen Fällen soll das System Hardwarefehler nicht selbst bearbeiten. Wird beispielsweise eine Anwendung für nicht unterstützte oder *Remote*-Operationen (Netzwerkserver) entwikkelt, soll vielleicht eine Nachricht über den Hardwarefehler an den *Remote*-Adressaten geschickt werden. Eine Meldung auf einem Bildschirm, die niemand liest, ist hier wenig sinnvoll. Ein weiteres Beispiel ist ein Datenbankverwaltungsprogramm, das die Integrität der Hauptdatenbank überwacht. Auch hier muß dem Programm über Hardwarefehler Bescheid gegeben werden, damit bestimmte Aktionen oder beispielsweise die Verwendung einer alternativen Hauptdatei auf einer anderen Einheit eingeleitet werden können. OS/2 erlaubt die Unterbindung einer automatischen Hardwarefehlerbehandlung auf einer Datei- oder Prozeßbasis. Im Netzwerkbeispiel soll die Bearbeitung von Hardwarefehlern beispielsweise für alle Prozesse unterbunden werden, im Datenbankbeispiel soll dies nur für die Hauptdatenbankdatei der Fall sein. Beim Auftreten eines Hardwarefehlers

in Verbindung mit einem Prozeß oder einer Dateinummer, wobei die automatische Hardwarefehlerbearbeitung desaktiviert ist, nimmt OS/2 eine FAIL-Antwort an und kehrt mit einem besonderen Fehlercode zur Anwendung zurück. Die Anwendung muß den Code analysieren und entsprechende Aktionen vornehmen.

Kapitel 18

Ein-/Ausgabeprivilegmechanismus und Debugging/Ptrace

In den vorangegangenen Kapiteln wurden die wesentlichen Architekturelemente von OS/2 dargestellt, die die Hauptleistungsmerkmale des Betriebssystems ausmachen. In einem Betriebssystem gibt es jedoch auch eine Vielzahl von "nebensächlichen Leistungsmerkmalen", ohne die die Effizienz des Gesamtsystems nicht gewährleistet wäre.

18.1 Ein-/Ausgabeprivilegmechanismus

Es wurde bereits auf die Notwendigkeit hingewiesen, Anwendungen einen direkten (und damit schnellen) Zugriff auf Geräteeinheiten zu gestatten. Dabei muß gewährleistet sein, daß erstens die Systemstabilität nicht gefährdet wird und zweitens mehrere Anwendungsprogramme nicht kollidieren. OS/2 verfügt für den direkten Einheitenzugriff über den *Ein-/Ausgabeprivilegmechanismus*. Dieses Leistungsmerkmal erlaubt einem Prozeß den direkten Zugriff auf Ein-/Ausgabeports einer Geräteeinheit (über den Einheitentreiber); auch auf Speicherstellen kann zugegriffen werden, die bestimmte Aufgaben besitzen oder in bestimmten Formaten belegt sind. Der Ein-/Ausgabeprivilegmechanismus kann von einer Anwendung direkt eingesetzt werden, was ihn notwendigerweise einheitenunabhängig macht. Die indirekte Anwendung durch ein Dynlink-Paket ist ebenfalls möglich. Das Dynlink-Paket kann dabei wie eine Art Einheitentreiber funktionieren; eine neue Implementation kann mit neuer Hardware zur Erhaltung der Kompatibilität eingesetzt werden. Der Pseudoeinheitentreiber ist üblicherweise wesentlich schneller als ein echter Einheitentreiber, weil eine angepaßte prozedu-

rale Schnittstelle verwendet wird. Da der Einsprung in Ring 0 und die Ausführung von OS/2-Kernel-Programmcode nicht benötigt werden, erspart dies viel Zeit.

Es gibt allerdings auch eine Einschränkung. Dynlink-Pseudoeinheitentreiber können alle Aufgaben ausführen, die auch ein echter Einheitentreiber ausführen kann, bis auf eine Ausnahme: die Bearbeitung von Interrupts. Da Hardwareinterrupts in Ring 0 bearbeitet werden, muß der bearbeitende Programmcode Teil eines echten Einheitentreibers ein. Häufig führt dies zu einem Kompromiß: Sowohl das Dynlink-Paket als auch ein echter Einheitentreiber werden bereitgestellt. Der Einheitentreiber behandelt die Interrupts, das Dynlink-Paket erledigt die restlichen Aufgaben. Ein Beispiel dafür stellt das KBD-Dynlink-Paket des Systems dar. Das VIO-Paket benötigt keinen Einheitentreiber zur Bearbeitung von Interrupts, da die Anzeigeeinheit keine Interrupts erzeugt.

Die beiden Komponenten des Ein-/Ausgabemodells von OS/2 sind der Zugriff auf den Einheitenspeicher und der Zugriff auf Ein-/Ausgabeports. Die Vergabe und Kontrolle des Zugriffs auf einen Memory-Mapped-Bereich einer Einheit ist einfach, da der Protect-Modus des 80286 über entsprechende Möglichkeiten zur Speicherverwaltung verfügt. Zunächst fragt ein Prozeß über den Einheitentreiber nach dem Zugriff auf den Speicher einer Einheit. Dies kann beispielsweise der Speicherpuffer einer Farb-/Graphikkarte sein. Üblicherweise führt dies ein Dynlink-Paket (statt der Anwendung) über einen *DosDevIOCtl*-Aufruf aus. Falls der Einheitentreiber die Anforderung bejaht, wird OS/2 über die *DevHlp*-Schnittstelle zur Initialisierung eines LDT-Speicherdeskriptors für die entsprechende physikalische Speicherstelle veranlaßt. OS/2 liefert als Resultat einen Selektor an den Einheitentreiber, der diesen an den aufrufenden Prozeß weitergibt. Diese Technik ist auf *Memory-Mapped*-Einheitenspeicher beschränkt. Geräteeinheitentreiber können sie verwenden, um Dynlink-Paketen den direkten Zugriff auf ein Datensegment des Einheitentreibers zu erlauben. Auf diese Weise kann die Verknüpfung von Dynlink-Paket und Einheitentreiber die Kommunikation an der Einheitenschnittstelle optimieren.

Die Bereitstellung des Zugriffs auf die Ein-/Ausgabeports ist schwieriger, da dieses Leistungsmerkmal vom 80286-Prozessor kaum unterstützt wird. Der 80286 verwendet einen Ringschutzmechanismus zur Steuerung des Ein-/Ausgabezugriffs. Das System kann einem gerade ausgeführten Programmcode mit einem bestimmten Ringprivileg Zugriff auf alle Ein-/Ausgabeports oder auf keinen Ein-/Ausgabeport gewähren. Einer Anwendung Zugriff auf alle Ports zu gewähren, ist gefährlich, da das VIO-Paket verwendet wird, und dieses den direkten Portzugriff auf den Anzeigeadapter benötigt. Das bedeutet, daß der Ein-/Ausgabebereich von OS/2 nicht geschützt ist, da nahezu alle Programme VIO oder den Presentation Manager direkt oder indirekt ansprechen.

Statt dessen wurde OS/2 so entwickelt, daß nach einer Anforderung an den Einheitentreiber alle Codesegmente Ein-/Ausgabezugriff besitzen, die in Ring 2 ausgeführt werden. Dies wird durch einen Befehl an den Linker erreicht. Der Nachteil dabei ist, daß der Zugriff auf Ein-/Ausgabeports uneingeschränkt gegeben werden muß. Ein Vorteil besteht darin, daß das System für Programmfehler nur anfällig ist, wenn diese bei der Ausführung von Ring-2-Segmenten auftreten. Die Leistungsmerkmale von Programmteilen, die in Ring 2 ausgeführt werden, sind beschränkt: Diese Programme können keine Dynlink-Aufrufe an das System ausführen. Hierfür gibt es zwei wesentliche Gründe:

- Die Unterstützung von Systemaufrufen in Ring 2 würde den Umfang von OS/2 unverhältnismäßig aufblähen.
- Es soll verhindert werden, daß Anwendungsprogrammierer ihre Anwendungsprogramme vollständig in Ring 2 ablaufen lassen, um Arbeit bei der Entwicklung der Ein-/Ausgaberoutinen zu sparen.

OS/2 Version 1.0 kann aufgrund des Ringmechanismus die Ein-/Ausgabe bis zu einem bestimmten Punkt schützen. Fehlerhafte Programme können die Systemstabilität durch die Manipulation der Peripherieeinheiten gefährden. Hinzu kommt, daß eine Real-Modus-Anwendung Ein-/Ausgabeanweisungen zu jedem

beliebigen Zeitpunkt ausführen kann. In künftigen OS/2- Versionen für den Prozessor 80386 wird dieses Problem gelöst sein. Die Hardware des 80386 erlaubt es, Prozessen den Zugriff auf einige (aber nicht alle) Ein-/Ausgabeports zu gewähren. Auf welche Ports zugegriffen werden darf, wird in einer Tabelle festgelegt, auf die das Anwendungsprogramm nicht zugreifen kann. Die Tabelle gilt sowohl im Protect- als auch im Real-Modus. Nebenbei: Der Real-Modus unter OS/2 ist eigentlich ein "virtueller Real-Modus", funktionell entspricht er aber dem Real-Modus früherer Prozessoren. OS/2 wird die vom Einheitentreiber bereitgestellten Portadressen verwenden, um den Zugriff auf bestimmte Ein-/Ausgabeports zu erlauben, die mit der Einheit kommunizieren. Der Prozeß wird darauf Zugriff haben. Durch diese Version werden Anwendungscodesegmente nicht unterstützt, die in Ring 2 ablaufen; alle so markierten Segmente werden geladen und in Ring 3 ausgeführt. Die Veränderung wird für Anwendungssoftware, die die korrekten Ein-/Ausgabeports verwendet, nicht erkennbar sein. Anwendungen, die den Zugriff auf eine Einheit anfordern und die Zugriffserlaubnis eigenmächtig auf andere Einheiten ausdehnen, werden erfolglos beendet werden.

18.2 Debugging/Ptrace

Da unter OS/2 sichergestellt sein muß, daß Anwendungen einander nicht beeinflussen, wurden besondere Leistungsmerkmale eingebaut, die Debuggingprogrammen die Manipulation und Untersuchung anderer Programme erlauben. Es gibt zu OS/2 einen Debugger, der vermutlich von den meisten Anwendungsprogrammierern eingesetzt wird. Nur wenige werden eigene Debugger entwickeln. Dennoch werden im folgenden die Grundsätze dargelegt, die Debugger erfüllen müssen, da dies einen weiteren Einblick in die Architektur von OS/2 ermöglicht.

Die erste Aufgabe eines Debuggers ist es, den Programmcode des zu bearbeitenden Programms lesen und schreiben zu können. Gleiches gilt für Datensegmente sowie für Signale, Unterbrechungspunkte und ähnliches. Alle diese Leistungsmerkmale

gehören zu OS/2, so daß das Betriebssystem die Informationen an den Debugger weiterreichen muß. Das zweite Ziel ist die Systemsicherheit: Eine Debuggerschnittstelle bietet für *Cracker*-Programme eine optimale Möglichkeit zur Manipulation von Programmen, indem Passworte und andere Schutzschemata umgangen werden. OS/2 verhindert dies, indem das zu untersuchende Programm bereits bei der Ausführung als solches markiert werden muß. Ein Debugger kann nicht in einen bereits ablaufenden Prozeß eingreifen. Hinzu kommt, daß Prozessen, die unter der Kontrolle eines Debuggers ablaufen, alle Zugriffsberechtigungen entzogen werden, die über die des Debuggers hinausgehen.

Die Ausführung von Kernel-Programmcode wurde bereits erläutert. Zusammenfassend läßt sich sagen, daß ein Thread, der eine Kernel-Anforderung ausgibt, selbst in den Kernel-Modus angehoben wird und dort seine eigene Anforderung bearbeitet. Dieses Prinzip vereinfacht die Kernel-Architektur, da eine Funktion zur Ausführung einer Aktion immer in einer seriellen und synchronen Art und Weise programmiert ist. Hinzu kommt, daß keine Zwänge auf einen Thread im Kernel-Modus ausgeübt werden können, jede im Kernel-Modus ausgeführte Aktion wird vom Thread selbst ausgelöst. Falls ein Prozeß beendet werden soll und einer seiner Threads sich im Kernel-Modus befindet, beendet OS/2 diesen Thread nicht; statt dessen wird eine Flagge (*Beendigung, sobald als möglich*) gesetzt, die den Thread beim Rücksprung in den Benutzermodus zur Beendigung veranlaßt. OS/2 benötigt keinen Programmcode zur Bestimmung und Freigabe interner Flaggen und Ressourcen, die von einem vorzeitig beendeten, im Kernel-Modus ablaufenden Thread zurückgeblieben sind. Zudem muß kein Thread den Status eines anderen Threads kennen oder verstehen. Der zu beendende Thread führt alle notwendigen Aktionen selbst aus, das heißt, er gibt Ressourcen und Flaggen frei und beendet sich anschließend selbst.

Wann ist für einen Thread der früheste Zeitpunkt zur Beendigung gekommen? Die Beendigung eines Threads ist ein erzwungenes Ereignis. Ein vom Kernel kommender Thread prüft direkt

vor dem Rücksprung in den Anwendungsmodus das Vorliegen eines erzwungenen Ereignisses ab. Diese Prüfung wird ständig ausgeführt (auch bei jeder Kontextumschaltung).

Obwohl es so scheinen mag, daß erzwungene Ereignisse lange Zeit unbearbeitet bleiben können, werden sie meist sehr schnell erkannt. Wenn beispielsweise ein Prozeß eine *DosKill*-Funktion für seine Kindprozesse ausgibt, wird jeder Thread des Kindprozesses für die vorzeitige Beendigung gekennzeichnet. Da dem Vaterprozeß die CPU-Arbeitszeit bei der Ausgabe des *DosKill*-Aufrufs zugeordnet war, befinden sich alle Threads des Kindprozesses im Kernel-Modus (da sie einen Systemaufruf abarbeiten oder wegen der Zuweisung der CPU-Zeit an einen anderen Prozeß, wobei die Threads in diesen Status versetzt werden). Bevor einer der nun markierten Threads auch nur eine einzige Anweisung des Anwendungsprogrammes abarbeiten kann, müssen sie durch die Verteilerroutine von OS/2. Die Flagge für die vorzeitige Beendigung wird bemerkt und die Threads beenden sich selbst, statt in den Anwendungsmodus zurückzuspringen. Es ist offensichtlich, daß diese Methode nicht zu langsam ist: Der *DosKill*-Aufruf tritt sofort in Kraft - nicht eine einzige Anweisung des Kindprozesses wird ausgeführt (mit Ausnahme der SIGKILL- und *DosExitList*-bearbeitenden Routinen). Die einzige signifikante Verzögerung in der Erkennung eines erzwungenen Ereignisses tritt auf, wenn die Bearbeitung eines Systemaufrufs viel Zeit in Anspruch nimmt. Da OS/2 nicht sehr CPU-gebunden ist, ist jeder Aufruf über einen "langen Zeitraum" (hier eine Sekunde oder länger) den größten Teil der Zeit blokkiert (in dieser Zeit findet keine Bearbeitung statt, sondern es wird auf ein Ereignis gewartet).

Wenn ein Kernel-Thread einen blockierenden Aufruf für ein Ereignis ausgibt, das erst nach langer Zeit auftreten kann (Warten auf einen Tastenanschlag oder auf die Freigabe einer Semaphore), wird eine Sonderform des blockierenden Aufrufs verwendet, der *unterbrechbare Blockierungsaufruf*. Sobald OS/2 die Flagge *Erzwungenes-Ereignis* für einen Thread setzt, wird geprüft, ob die Blockierung des Threads unterbrochen werden kann. Ist dies der Fall, wird die Blockierung aufgehoben und an

den Thread ein besonderer Code übergeben (*die Bearbeitung wird nicht aufgenommen, weil das erwartete Ereignis eintrat, sondern weil ein erzwungenes Ereignis entdeckt wurde*). Der Thread muß den Systemaufruf nun rasch beenden (üblicherweise durch die Deklaration eines Fehlers), so daß der Thread die Verteilerroutine durchlaufen und die gesetzte Flagge erkennen kann. Der hierzu verwendete Mechanismus wurde bereits für eine andere Art der erzwungenen Ereignisse erläutert - den Signalmechanismus von OS/2. Ein ankommendes Signal ist ein erzwungenes Ereignis für den Thread 1 eines Prozesses und besitzt dieselbe Antwortzeit wie der Abbruch eines "langsamen" Systemaufrufs.

Die umfangreiche Erläuterung der Behandlung erzwungener Ereignisse wurde eingefügt, da das interne Debugging-Leistungsmerkmal auf einem besonderen, erzwungenen Ereignis beruht. Beim Eintreten eines Prozesses in den Fehlersuchstatus wird seine *Trace*-Flagge für den initiierenden Thread des Prozesses permanent gesetzt. Gleiches gilt für jeden davon erzeugten Thread. Wenn einer dieser Threads im Kernel-Modus ist (in den Kernel-Modus wird immer dann gesprungen, wenn eine Aktion durchzuführen zu ist), wird der zu untersuchende Code (eine Hälfte der Fehlersuche) ausgeführt. Die Debugger-Hälfte wird von einem Debugger-Thread ausgeführt, der besondere *DosPtrace*-Aufrufe ausgibt. Die beiden Hälften des Paketes kommunizieren über einen gemeinsam benutzten Speicherbereich.

Wenn das zu untersuchende Programm ein außergewöhnliches Ereignis erkennt (zum Beispiel die Betätigung der Tastenkombination Ctrl-C oder einen GP-Fehler), hat das *Trace-Force*-Ereignis Vorrang über jedes andere Ereignis. Der Thread des zu untersuchenden Programmes führt seine Hälfte des *DosPtrace*-Codes aus. Der Code schreibt einen Datensatz in einen Kommunikationspuffer, der das Ereignis beschreibt, aktiviert den Debugger-Thread, der typischerweise im Debugger-Teil des *DosPtrace*-Codes blockiert ist und blockiert selbst, um auf eine Antwort zu warten. Der Debugger-Thread wird aktiv und kehrt mit der Ereignisinformation zum Debugger zurück. Wenn der

Debugger *DosPtrace* mit einem Kommando wieder aufruft, wird dieses in den Kommunikationsbereich geschrieben und das zu untersuchende Programm liest den Befehl und führt ihn aus. Kommandos können hier sein: *Normale Ausführung aufnehmen* oder *Wie ist der Inhalt bestimmter Speicherstellen*, woraufhin der Thread des zu untersuchenden Programms antwortet und im *DosPtrace*-behandelnden Programmteil verbleibt.

Dies ist einfach zu implementieren, funktioniert und nutzt die neuen OS/2-Leistungsmerkmale. So wird beispielsweise kein Programmteil zum Zugriff auf den Adreßbereich des zu untersuchenden Programmes verwendet, da das zu untersuchende Programm selbst (unwissentlich im *DosPtrace*-Code) den eigenen Adreßbereich liest und schreibt. Dieses Leistungsmerkmal ist der *ptrace*-Utility von UNIX nachempfunden.

Im folgenden finden Sie einige Punkte, die Programmierern bei der Verwendung der Debugging-Möglichkeiten bekannt sein sollten. OS/2 unterhält eine Verknüpfung zwischen Debugger und zu untersuchendem Programm. Bei der Beendigung des Debuggers wird das bearbeitete Programm ebenfalls beendet (falls dies noch nicht der Fall war). Das zu untersuchende Programm muß kein direkter Kindprozeß des Debuggers sein. Wenn der Debugger den anfänglichen *DosPtrace*-Aufruf ausführt, erstellt OS/2 eine Verbindung zwischen Debugger und dem letzten Programm, das mit der Debugging-Option ausgeführt wurde. Bei der Ausführung eines Prozesses mit Debugging-Option ohne nachfolgenden *DosPtrace*-Aufruf wird das ausgeführte Programm nach zwei Minuten beendet.

Kapitel 19

Privilegierte Funktionen und die 3X-Box

Als die Entwicklung von OS/2 gestartet wurde, war von Anfang an Voraussetzung, daß bestehende MS-DOS-Anwendungen problemlos unter dem neuen Betriebssystem laufen mußten. Es wurde allerdings auch sehr schnell offensichtlich, daß die Erfüllung dieser Anforderung wie auch die Umsetzung nicht einfach ist.

Die Bereitstellung der typischen MS-DOS-Schnittstellen unter OS/2 und die Sicherung der Aufwärtskompatibilität auf diese Weise wäre einfach. Leider würden jedoch die meisten populären Anwendungsprogramme in einer solchen Umgebung nicht funktionieren. Hochentwickelte MS-DOS-Anwendungen übernehmen durchweg die direkte Kontrolle der Umgebung und verwenden MS-DOS nur für ausgewählte Aufgaben wie etwa die Dateiverwaltung oder Tastaturpufferung. Damit MS-DOS-Anwendungen unter OS/2 ablaufen können, muß also eine vollständige DOS-ähnliche Real-Modus-Umgebung geschaffen werden, die einfache Unterstützung des INT 21 genügt nicht.

OS/2 bietet eine DOS-kompatible Umgebung, die *Real-Modus-Bildschirmgruppe*, *Kompatibilitätsbox* oder einfach *3X-Box* genannt wird. Die 3X-Box emuliert die Umgebung eines PC auf der Basis des 80286 unter MS-DOS Version 3.3 (für die 3X-Box von OS/2 Version 1). MS-DOS-Programme werden im Real-Modus ausgeführt. Da die Emulation des Real-Modus aus dem Protect-Modus heraus sehr langsam ist, schaltet OS/2 zur Ausführung von MS-DOS-Anwendungen physikalisch in den Real-Modus um. MS-DOS-Programme arbeiten mit der Speicherbelegung von MS-DOS, daher wird diese in der 3X-Box von OS/2

ebenfalls emuliert. Die ersten *n* Kbytes (üblicherweise 640 Kbytes) des unteren Speicherbereichs werden ausschließlich für den Einsatz bestimmter OS/2-Teile und für die 3X-Box reserviert. Protect-Modus-Anwendungen verwenden diesen Speicherbereich nie. Dadurch können Programme, die für ein Singletasking-System geschrieben sind, in einer Multitasking-Umgebung ausgeführt werden. In Bild 19.1 ist die Speicheraufteilung von OS/2 dargestellt. Die unteren 640 Kbytes des Speicherbereichs sind für Einheitentreiber und Teile von OS/2 reserviert, die im Real-Modus ablaufen. Der Rest des Bereiches ist der 3X-Box zugeordnet. Der Speicherbereich von 640 Kbytes bis zu einem Mbyte ist für das ROM und den Bildschirmpuffer vorgesehen. Der darüber hinausgehende Speicher dient der Aufnahme weiterer Systemteile von OS/2; hier finden auch alle im Protect-Modus ablaufenden Anwendungen ihren Platz. Nicht auslagerbare Segmente werden an einem Ende des Speicherbereichs gehalten, um die Speicheraufsplittung zu minimieren.

OS/2 verwendet den Bildschirmgruppenmechanismus zur Bereitstellung einer Benutzerschnittstelle für die 3X-Box. Eine Bildschirmgruppe wird von OS/2 automatisch als *Real-Modus-Bildschirmgruppe* definiert. Beim ersten Aktivieren der Bildschirmgruppe führt OS/2 COMMAND.COM aus. Der Benutzer hat durch die Anwahl der Bildschirmgruppe den Zugriff auf die Real-Modus-Umgebung und kehrt durch die Wahl einer anderen Bildschirmgruppe in den Protect-Modus zurück. OS/2 Version 1.0 unterstützt nur eine einzige Bildschirmgruppe für den Real-Modus, da die Real-Modus-Kompatibilität durch die tatsächliche Umschaltung in den Real-Modus realisiert wird. Daher ist nur ein einziger 640-Kbyte-Bereich für Real-Modus-Anwendungen vorgesehen. Die Unterstützung mehrerer Real-Modus-Anwendungen ist beim 80286-Mikroprozessor schwierig. Der 80386 hingegen bietet als besonderes Hardware-Leistungsmerkmal den *virtuellen 8086-Modus*, wodurch künftigen OS/2-Versionen die Unterstützung mehrerer Real-Modus-Bildschirmgruppen erlaubt werden (sofern das Betriebssystem auf einem 80386 installiert ist).

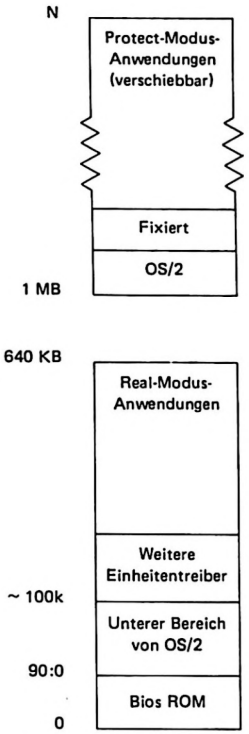


Bild 19.1 Speicheraufteilung des Systems

Das Betriebssystem, das in der 3X-Box die INT 21-Anforderungen bearbeitet, ist nicht mit MS-DOS identisch. Vielmehr stellt es eine Erweiterung von OS/2 im unteren Speicherbereich dar. Da die Wurzeln von OS/2 in MS-DOS liegen, führen beide Betriebssysteme Funktionsaufrufe in ähnlicher Weise aus. OS/2 unterstützt Funktionen (die keine MS-DOS-Funktionen sind), indem es möglichst wenig eingreift und die 3X-Anwendung die Hardware nach eigenem Willen verwenden läßt. So ist beispielsweise das Verbiegen der meisten Interruptvektoren gestattet (etwa des INT 21 und der Interruptvektoren des ROM-BIOS).

ROM-BIOS-Aufrufe werden ohne Einschränkungen unterstützt. Da OS/2 Real-Modus-Aufrufe überwachen muß, die an Treiber ausgegeben werden, kann die Einmischung des Betriebssystems nicht so einfach (wie dies scheinen mag) verhindert werden. Dies ist nötig, um Konflikte mit asynchronen, im Protect-Modus ablaufenden Disketten- bzw. Plattenein-/-ausgaben zu vermeiden. Für OS/2 kann es nötig werden, eine Real-Modus-Anwendung temporär zu blockieren, weil ein BIOS-Aufruf erst ausgeführt werden kann, wenn der Einheitentreiber im Protect-Modus die Hardware freigegeben hat. Nachdem die Real-Modus-Anwendung sich im BIOS befindet, schützt derselbe Mechanismus den Einheitentreiber im Protect-Modus vor dem Einsprung in einen kritischen Bereich.

Hardwarefehler werden von Real-Modus-Anwendungen durch den OS/2-Hardwarefehler-Daemon und den INT-24-Mechanismus der 3X-Box behandelt:

- *Schritt 1*: Bei Hardwarefehlern, die nur in der OS/2-Umgebung auftreten, wird der Hardwarefehler-Daemon aktiviert. Dieser teilt dem Benutzer das aufgetretene Problem mit und speichert die Benutzerantwort. Bei Hardwarefehlercodes, die auch unter MS-DOS Version 3.3 verfügbar sind, wird dieser Schritt übersprungen und als erstes der zweite Schritt abgearbeitet.

- *Schritt 2*: Hat die Real-Modus-Anwendung eine eigene fehlerbehandelnde Routine über INT 24 installiert, wird diese aufgerufen. Falls Schritt 1 übersprungen wurde, sollte der zurückgegebene Code der Anwendung bekannt sein und wird daher unverändert übergeben. Bei vorheriger Ausführung von Schritt 1 wird der Fehlercode in einen allgemeinen-INT-24-Fehler umgewandelt. Die vom Programm gelieferte Antwort, soweit für diesen Fehlertyp gültig, wird ausgeführt. Dadurch werden sowohl Benutzer als auch Anwendung vom Auftreten einer Fehlersituation unterrichtet. Anschließend wird die vom Programm angedeutete Fehlerbehandlung (und nicht die vom Benutzer angegebene) durch den Hardwarefehler-Daemon vorgenommen.

- *Schritt 3*: Falls die Real-Modus-Anwendung keine fehlerbehandelnde Routine über INT 24 gesetzt hat, stellt OS/2 eine fehlerbehandelnde Standardroutine bereit. Wurden Schritt 1 und der Hardwarefehler-Daemon bereits ausgeführt, erledigt OS/2 nun die durch den Benutzer spezifizierte Lösung. Falls der aufgetretene Fehler MS-DOS bekannt ist, die Real-Modus-Anwendung aber keine fehlerbehandelnde Routine installiert hat, wird nun der Hardwarefehler-Daemon aufgerufen, um eine Meldung anzuzeigen und eine Benutzereingabe entgegenzunehmen.

Die 3X-Box unterstützt nur MS-DOS-Funktionalität, neue OS/2-Leistungsmerkmale sind in 3X-Box-Anwendungen nicht verfügbar (keine API, keine mehrfachen Threads, kein IPC, keine Semaphoren und ähnliches). Dies hat zwei Gründe: Obwohl grundsätzlich jede Real-Modus-Anwendung die Systemstabilität zerstören kann, macht der Zugriff auf Leistungsmerkmale des Protect-Modus dieses Problem noch gravierender. Das Beenden und im Speicher verbleiben eines Programmes durch die Manipulation der CPU kann dazu führen, daß eine Real-Modus-Anwendung einen kritischen Bereich mit einer Semaphore schützt, aber nicht garantiert, daß die Semaphore wieder freigegeben wird. Zum zweiten besitzt OS/2 nur eine Real-Modus-Box, die unterhalb der 640-Kbyte-Grenze arbeitet. Es ergibt keinen Sinn, neue Real-Modus-Anwendungen zu entwikkeln, die OS/2-Funktionen verwenden, den Protect-Modus aber nicht nutzen.

Die Emulation der 3X-Box gilt auch für Interrupts. OS/2 fährt mit der Umschaltung der CPU auch fort, wenn die 3X-Box aktiv ist (das heißt, die Anwendung der 3X-Box ist die Vordergrundanwendung). Der Prozeß wird allerdings nur dann vorzeitig abgebrochen, wenn eine zeitkritische, im Protect-Modus ablaufende Anwendung CPU-Zeit benötigt, oder wenn die Real-Modus-Anwendung blockiert. Führt die CPU beim Eintreffen eines Einheiteninterrupts eine Anwendung im Protect-Modus aus, wird in den Real-Modus umgeschaltet, so daß die Anwendung im Real-Modus (die den Interruptvektor verbiegt) den Interrupt auch im Real-Modus empfangen kann. Nach Beendi-

gung des Interrupts schaltet OS/2 in den Protect-Modus zurück und führt die im Protect-Modus laufende Anwendung weiter aus.

Anwendungsprogramme im Protect-Modus können im Hintergrund weiterarbeiten, wenn sich die 3X-Box im Vordergrund befindet. Der umgekehrte Fall ist nicht möglich. Liegt die 3X-Box im Hintergrund vor, wird die Ausführung des Programmes einschließlich der Interrupts unterbrochen. Im Gegensatz zu Protect-Modus-Anwendungen läßt sich bei Real-Modus-Anwendungen nicht sicherstellen, daß diese nicht die Bildschirmhardware aus dem Hintergrund heraus beeinflussen. Eine Real-Modus-Anwendung bemerkt ihre Unterbrechung im Hintergrund üblicherweise nicht; das einzige, woran sie eine Unterbrechung erkennen kann, ist der "plötzliche Sprung" der Tageszeit. Da die Umschaltung zwischen den Modi ein langsamer Prozeß ist und die Interrupts für nahezu eine Millisekunde unterbunden werden, kann dies bei schnellen Einheiten wie seriellen Ports zu Überläufen führen. Am besten wird die Anwendung im Real-Modus in eine Hintergrundbildschirmgruppe geschaltet; falls keine Real-Modus-Programme auszuführen sind, beendet OS/2 auch die Umschaltung.

Einige OS/2-Utility-Programme (wie FIND) sind als Family-API-Anwendung implementiert. Ein einziges Binärprogramm kann sowohl im Protect- als auch im Real-Modus ablaufen, dem Benutzer wird die Unbequemlichkeit des Hin- und Herschaltens vom Real- in den Protect-Modus bei der Ausführung von Hilfsfunktionen erspart. Dies funktioniert für einfache Utility-Programme ohne bildschirmorientierte oder graphische Schnittstellen und für Programme, die keine übertriebenen Speicheranforderungen haben und auch sonst nur wenig von den erweiterten OS/2-Leistungsmerkmalen Gebrauch machen. Anwendungen, die sich intensiv auf OS/2-Funktionalität abstützen, sollten im Protect-Modus entwickelt werden.

Kapitel 20

Family API

Bei der Ankündigung einer neuen Version eines PC-Betriebssystems stehen die Entwickler von Anwendungsprogrammen vor einer Entscheidung: Sollen die Programme die neuen Leistungsmerkmale nutzen oder soll nur auf die Funktionalität früherer Versionen zurückgegriffen werden? Durch die Verwendung der neuen Merkmale könnte ein Produkt "besser" im Sinne von "leistungsfähiger" werden. Hinzu kommt, daß die Programmierung üblicherweise einfacher ist. Wenn ein solches Programm auf den Markt kommt, muß man jedoch davon ausgehen, daß nur etwa ein Zehntel der installierten PCs mit der neuen Version arbeiten. Nicht alle existierenden Systeme verfügen über den Prozessor, der für die Verwendung des neuen Betriebssystems notwendig ist. Zudem sehen viele Benutzer zunächst keine Notwendigkeit für die Anschaffung der neuen Betriebssystemversion. Falls ein Anwendungsprogramm so entwickelt werden kann, daß es nur die Leistungsmerkmale der bisherigen Betriebssystemversionen nutzt (und unter dem neuen Betriebssystem in einem Kompatibilitätsmodus abläuft), ist die Versuchung zur Wahl dieser Lösung groß. Selbst wenn dadurch Einschränkungen bei der Leistungsfähigkeit des Produktes hingenommen werden müssen, ist der potentielle Markt zehnmal so groß im Vergleich mit einem Produkt, das nur auf dem neuen Betriebssystem läuft.

Dies ist der klassische Teufelskreis bei Softwarestandards: Wenn der Benutzer keine Notwendigkeit darin sieht, wird er das neue System nicht benutzen. Falls das neue System nicht benutzt wird, entwickeln Programmierer keine Anwendungsprogramme dafür und daher wird der Benutzer niemals die Notwendigkeit

der Benutzung des neuen Betriebssystems erkennen. Es dauert folglich sehr lange, bis für ein neues Betriebssystem Anwendungen bereitstehen.

In OS/2 wird dieses Problem auf mehrere Arten behandelt. Die mit OS/2 ausgelieferte Software läuft im Protect-Modus. OS/2 enthält zusätzlich möglichst viele Benutzerfunktionen, um den Wert des Betriebssystems für einen Benutzer zu steigern, der keine Anwendungen im Protect-Modus besitzt. Das wichtigste neue Leistungsmerkmal von OS/2, das zur Benutzerakzeptanz beitragen wird, ist die *Family API*.

Die Family API stellt eine Teilmenge der API von OS/2 im Protect-Modus dar. Unter Verwendung der im Developer's Kit von OS/2 bereitgestellten Tools können Anwendungen entwikkelt werden, die die Family API nutzen. Die resultierenden .EXE-Dateien laufen ohne Veränderung unter OS/2 im Protect-Modus oder mit einem 8086 Prozessor unter dem Betriebssystem MS-DOS 2.X oder 3.X. Selbstverständlich funktionieren solche Anwendungen auch in der 3X-Kompatibilitätsbildschirmgruppe unter OS/2. Allerdings sollte die einzige Real-Modus-Bildschirmgruppe, die es in OS/2 gibt, nicht für ein Anwendungsprogramm "verschwendet" werden, das auch im Protect-Modus funktioniert. Entwickler müssen sich also nicht entscheiden, ob eine Anwendung unter OS/2 im Protect-Modus oder für das Betriebssystem MS-DOS geschrieben werden soll, sie verwenden einfach die Family API und halten sich somit beide Möglichkeiten offen. Die Programme laufen als Protect-Modus-Anwendung unter OS/2 oder als MS-DOS-Anwendung unter einem MS-DOS-System.

Ohne Zweifel handelt es sich bei der Family API um eine bemerkenswerte Einrichtung. Sie bietet OS/2-Funktionalität mit einer Dynlink-Schnittstelle zu Programmen, die ohne OS/2 unter MS-DOS laufen. Realisiert wird dies durch die OS/2-Kompatibilitätsbibliothek, die OS/2-Systemschnittstellenaufrufe akzeptiert und diese selbst implementiert (der zugrundeliegende MS-DOS-Interrupt 21 wird nach Bedarf aufgerufen). Es stellt sich die Frage, welche Funktionen in die Family API eingebun-

den sind: natürlich alle Funktionen, die entsprechende INT-21-Funktionen besitzen - wie *DosOpen*, *DosRead* und *DosAllocSeg*. Ebenfalls verfügbar sind Funktionen wie *DosSubAlloc*, die direkt durch eine dazu bestimmte Family-API-Bibliothek unterstützt werden. Leistungsmerkmale, die in einer echten MS-DOS-Umgebung extrem schwer zu unterstützen sind (wie die Verwaltung mehrerer Threads und die asynchrone Ein- und Ausgabe), sind in der Family API nicht verfügbar.

Woher stammt die Bibliothek und wie wird sie von MS-DOS geladen, um Dynlink-Anforderungen auszuführen? Dies wird durch sogenannte *Spiegel* verwirklich, die in der .EXE-Datei enthalten sein müssen (bei der Ausführung einer Family-API-Anwendung unter MS-DOS ist nur diese Datei verfügbar). In Bild 20.1 wird das Gerüst einer .EXE-Datei gezeigt, die die Family API nutzt.

Für OS/2 mußte eine neue Struktur für die .EXE-Datei definiert werden, da die Informationen in den .EXE-Dateien von MS-DOS nun nicht mehr ausreichend sind. Da die Speicherarchitektur des 8086 linear ist, beschreiben .EXE-Dateien unter MS-DOS lediglich einen Speicherbereich, der zusammenhängend geladen wurde. Für OS/2 muß jedes Segment separat beschrieben werden. Dabei müssen Informationen über den Segmentstatus (Nur-Lesen, Programm- oder Datensegment usw.) angegeben werden. Auch müssen im .EXE-Format unter OS/2 bestimmte Datensätze enthalten sein, die die dynamischen Verknüpfungen beim Ladevorgang beschreiben. Das neue .EXE-Format wurde so festgelegt, daß es nach außen hin genau dem Header einer .EXE-Datei unter DOS entspricht. In diesem Header wird eine Flagge gesetzt, die von allen MS-DOS-Versionen ignoriert wird. OS/2 erkennt daran, daß dies eine .EXE-Datei im OS/2-Format und keine MS-DOS-Datei ist.

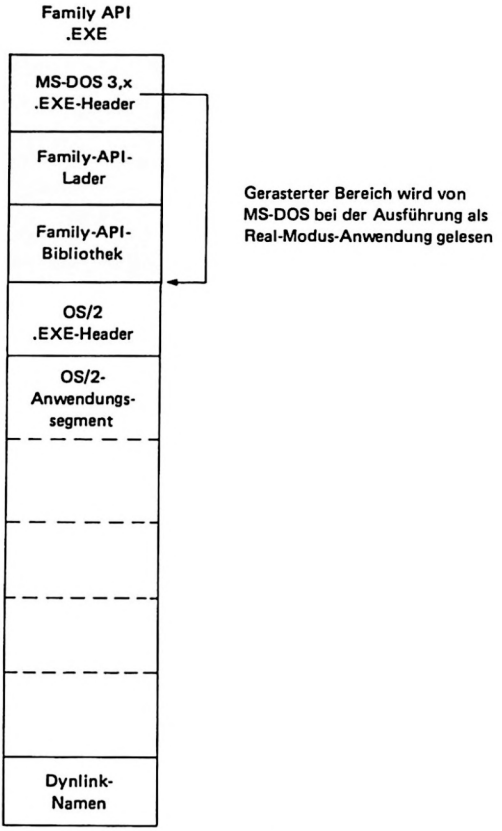


Bild 20.1 .EXE-Dateiformat einer ausführbaren Family-API-Anwendung

Beim Laden der .EXE-Datei auf einem MS-DOS-System wird der Header im alten Format verwendet. Er beschreibt nicht die Anwendung selbst, sondern einen Programmteil in der .EXE-Datei, der vor der eigentlichen Anwendung gespeichert ist: den Family-API-Lader und die Bibliothek. Für MS-DOS sieht die .EXE-Datei wie ein ausführbares Programm aus, bei dem Fami-

ly-API-Lader und Bibliothek das Programm darstellen. Lader und Bibliothek werden in den Speicher geholt und die Ausführung beginnt. MS-DOS lädt die Anwendung selbst nicht, da dies nicht im .EXE-Header beschrieben war. Sobald die Ausführung beginnt, liest der Family-API-Lader die Anwendungssegmente, führt Relokationsberechnungen aus und erfüllt Dynlink-Referenzen mit entsprechenden Einsprungpunkten im Bibliothekspaket der Family API. Nach dem Laden verschiebt der Family-API-Lader die Anwendung zur endgültigen Ausführungsadresse, wobei ein Großteil des Family-API-Laders überlagert und die Anwendung ausgeführt wird. Alle ausführbaren OS/2-Dateien besitzen diesen MS-DOS-Header. Falls eine nicht in der Family API ausführbare Datei vorliegt, sind der Family-API-Lader und die Bibliothek nicht vorhanden und der Header beschreibt standardmäßig eine unmöglich große, ausführbare MS-DOS-Datei. Sollte eine solche Anwendung irrtümlicherweise unter einem System, das nicht OS/2 ist, oder in der OS/2-Kompatibilitätsbildschirmgruppe ausgeführt werden, wird MS-DOS das Programm nicht laden. Optional kann der Programmierer ein kleines Programm einbinden, das mit dem Family-API-Lader geliefert wird und eine Fehlermeldung auf dem Bildschirm ausgibt. Der althergebrachte .EXE-Header am Beginn einer Datei enthält eine Flagge, die OS/2 auf die Existenz eines .EXE-Headers der neuen Architektur in der Datei hinweist. Da der Header nicht den Family-API-Lader und die Bibliotheksteile der Datei beschreibt, ignoriert OS/2 deren Vorhandensein und lädt die Family-API-Anwendung im Protect-Modus. Die Dynlink-Referenzen werden mit den normalen Dynlink-Bibliotheken ausgeführt und die Family-API-Versionen nicht genutzt.

Selbstverständlich sind Dynlink-Aufrufe in einer MS-DOS-Umgebung keine echten dynamischen Verknüpfungen, sondern werden nur als solche vorgetäuscht. Die dynamische Verknüpfung zur Laufzeit wird nicht unterstützt, wohl aber die Verknüpfung mit Zielbibliotheken beim Laden (diese werden in die .EXE-Datei eingebunden). Einer der Vorteile von dynamischen Verknüpfungen ist, daß der Zielcode nicht Teil der .EXE-Datei ist und daher ohne Änderung der .EXE-Datei beliebig verän-

dert oder aktualisiert werden kann. Dies gilt nicht für die Dynlink-Emulationsbibliothek der Family API, da diese tatsächlich in die .EXE-Datei eingebunden ist. Üblicherweise wird dieser Nachteil nicht zum Problem. Dynlink-Bibliotheken werden entweder zur Verbesserung ihrer Implementation oder zum Hinzufügen neuer Leistungsmerkmale aktualisiert. Die Family-API-Bibliothek kann nicht mehr wesentlich verbessert werden, da deren Umgebung (MS-DOS) sich künftig nicht mehr verändern wird. Werden neue Family-API-Leistungsmerkmale hinzugefügt, so hätte das Laden der neuen Bibliothek mit bereits existierenden .EXE-Dateien, die von der Family API Gebrauch machen, keine Bedeutung; diese Programme nutzen die neuen Leistungsmerkmale sowieso nicht.

Einen schwerwiegenden Nachteil stellen die Größe der Familiy API und die Ladegeschwindigkeit der Familiy-API-Anwendungen dar. Es ist offensichtlich, daß die Größe einer .EXE-Datei unter Verwendung der Family API anwächst, da der Family-API-Lader und die unterstützten Bibliotheken eingebunden werden müssen. Die Tools zur Erstellung einer ausführbaren Datei mit der Family API binden nur Bibliotheksroutinen ein, die vom Programm benutzt werden. Aber auch so sind Bibliothek und Lader relativ umfangreich (üblicherweise zehn bis 14 Kbyte in der .EXE-Datei und neun Kbyte - ohne Lader - im RAM). Zudem ist der Ladevorgang einer Family-API-Anwendung unter MS-DOS langsamer als das Laden einer echten MS-DOS-Datei. Die ausführbare Datei unter MS-DOS besteht aus einem einzigen zusammenhängenden Block von Daten, die in einer Datenträgerleseoperation in den Speicher geladen werden. Hinzu kommt eine meist wenig umfangreiche Relokationstabelle. Eine Family-API-Anwendung benötigt zum Laden mehr Zeit, da zuerst der Lader und die Bibliotheken geladen werden müssen und die eigentliche Anwendung erst danach segmentweise in den Speicher gebracht wird.

Der Unterschied zwischen MS-DOS 3.X und OS/2 kann nicht vollständig verborgen werden, obwohl die Family API die Entwicklung von Anwendungen ermöglicht, die in beiden Umge-

bungen ablauffähig sind. So unterstützt die Family API zu einem Zeitpunkt nur die *DosFindFirst*-Funktion für eine einzige Suchnummer. Eine Anwendung, die mehrere Suchläufe zur gleichen Zeit ausführen will, sollte *DosGetMachineMode* zur Bestimmung der Umgebung verwenden. Anschließend kann der *DosFindFirst*-Aufruf im Protect-Modus erfolgen oder ein Aufruf des INT 21 im Real-Modus. Auch Anwendungen zur Beeinflussung von Druckerdaten benötigen versionsspezifische Programmteile zum Verbiegen des Interrupt 17 oder zur Verwendung von Einheitenmonitoren (abhängig von der Umgebung).

Teil 3
Die Zukunft

Kapitel 21

Die Zukunft

Die Darstellung in diesem Kapitel ist nicht einfach, da die Beschreibung einer neuen OS/2-Version bei Freigabe uninteressant wird. Die Vorankündigung von Produkten kann ebenfalls nicht Sinn dieses Kapitels sein, da Abweichungen von der Zeitplanung und Kundenforderungen die Einbeziehung bestimmter Leistungsmerkmale beschleunigen können, während andere Merkmale weggelassen oder einer späteren Version vorbehalten werden. Bei einer expliziten Beschreibung zukünftiger Leistungsmerkmale könnten Entwickler ihre Programme so planen, daß sie noch nicht vorhandene Möglichkeiten einbeziehen, was zu Problemen führt, wenn die Implementierung eines Leistungsmerkmals zurückgestellt oder gar ganz aus der Planung gestrichen wird. In diesem Kapitel wird deshalb nicht die Funktionalität künftiger OS/2-Versionen beschrieben, sondern nur die Zielvorstellung. Entwickeln Sie Ihre Anwendungen so, daß sie nicht von Leistungsmerkmalen abhängig sind, die in künftigen OS/2-Versionen unterstützt werden.

Bei OS/2 Version 1.0 handelt es sich um das erste zum Standard-MS-DOS kompatible Betriebssystem, das das Speicheradressierungspotential des 80286 verfügbar macht. Daher wird es einerseits viele APIs geben, die diesem Standard folgen. Auf der anderen Seite können vorhersehbare künftige Versionen keine große Marktdurchdringung erwarten. Die Entwickler von OS/2 Version 1.0 haben sich daher in erster Linie *auf die Verfügbarkeit der APIs* konzentriert. Die Durchsatzsteigerung wurde auf zukünftige Versionen verschoben, da diese Art der Verbesserung leichter realisiert werden kann als das Hinzufügen

neuer APIs. Für die absehbare Zukunft ist geplant, die existierenden Schnittstellen auszubauen, nicht neue hinzuzufügen.

21.1 Dateisystem

Den Kern jeder Büroautomation stellen Daten dar. Die Suche nach Daten, das Lesen und natürlich auch das Schreiben von Daten werden in Zukunft zum wichtigsten Aspekt der Datenverarbeitung werden. Die reine Verarbeitungs- und Rechenleistung eines Systems ist in dieser Umgebung relativ uninteressant. Der wichtige Punkt ist, wie schnell ein System Daten erhalten und manipulieren kann. Sicherlich ist eine leistungsfähige CPU von Vorteil. Sie erlaubt beispielsweise die Verwendung rechenintensiver graphischer Benutzerschnittstellen. Der Durchsatz der Ein-/Ausgabe wird zum begrenzenden Faktor, insbesondere in der Multitasking-Umgebung. Woher kommen die Daten? Bei der Dateneingabe über die Tastatur werden sicherlich keine Probleme auftreten, da die Eingabegeschwindigkeit eines Menschen im Vergleich zur Verarbeitungsgeschwindigkeit eines Computers gering ist. Für andere Daten, die nicht von einem Massenspeicher kommen, sollten die OS/2-Einrichtungen für Direktzugriffseinheiten einen genügend großen Dateidurchsatz erlauben. Es bleibt die Frage nach dem Dateisystem für lokale Laufwerke und das Netzwerk für den Zugriff auf an anderer Stelle gespeicherte Daten (Remote-Zugriff). Das Dateisystem bietet sich für eine künftige Erweiterung an. Die Schnittstelle ist allgemein gehalten, so daß für die erste Version entwickelte Anwendungen auch mit künftigen Versionen ablauffähig sein werden.

Die Erweiterung des Dateisystems ist insofern relativ einfach, als daß die Schwachstellen des momentan genutzten FAT-Dateisystems offensichtlich sind.

- *Unterstützung umfangreicher Datenträger.* Ein neues Dateisystem wird beliebig umfangreiche Laufwerke ohne eine zu weitgehende Belegungsfragmentierung unterstützen. Die Fragmentierung resultiert aus der Tatsache, daß das Dateisystem mit einer kleinsten Belegungseinheit operieren muß.

Falls die Belegungseinheit die Größe *N* besitzt, kann davon ausgegangen werden, daß jede Datei im Mittel *N/2* Bytes der Datenträgerkapazität ungenutzt läßt. Der Grund: Jede Datei besitzt eine letzte Belegungseinheit, die im Mittel nur zur Hälfte benötigt wird. Bei einer sehr großen Belegungseinheit (beispielsweise größer als zwei Kbyte), ist der durchschnittliche Verlust aufgrund der Fragmentierung sehr groß, da viele Dateien kleiner als die Einheit sind.

Das bestehende FAT-Dateiverwaltungssystem von MS-DOS kann mit umfangreichen Datenträgern arbeiten, verschwendet dabei aber relativ viel Speicherkapazität durch die Verwendung großer Belegungseinheiten. Abhängig von der Anzahl und Größe der Dateien kann ein 100-Mbyte-Datenträger durch die Fragmentierung bis zu 50 % seiner Speicherkapazität verlieren. Das neue Microsoft-Dateiverwaltungssystem wird kleine Belegungseinheiten (beispielsweise 512 Bytes) unterstützen, um den Fragmentierungsverlust zu reduzieren. Dabei werden die kleinen Belegungseinheiten den Durchsatz des Dateisystems nicht negativ beeinflussen.

- *Schutz von Dateien.* Das neue Dateiverwaltungssystem muß den Schutz von Dateien unterstützen, um eine vollständig sichere Umgebung bieten zu können. Der Schutz von Dateien ist üblicherweise ein Leistungsmerkmal von Mehrbenutzersystemen. Das FAT-Dateiverwaltungssystem von MS-DOS wurde für eine Einbenutzerumgebung entwickelt und enthält folglich keine Leistungsmerkmale zum Schutz vor unberechtigtem Zugriff. Wozu wird der Schutz von Dateien benötigt? Ein Grund ist, daß ein vernetzter PC zwar physikalisch eine Einheit für einen Benutzer darstellt, logisch ist er aber Teil eines Mehrbenutzersystems, da mehrere Benutzer über das Netzwerk auf die gleichen Dateien zugreifen können. Oftmals ist es sogar zweckdienlich, eigene Dateien vor dem eigenen Zugriff zu schützen.

  Heutzutage bestehen die meisten Netzwerkinstallationen aus Server- und Klientengeräten, wobei Klientensysteme nur

auf Dateien zugreifen können, die auf Server-Systemen vorliegen. MS-Net- und PC-Net-Server besitzen eine rudimentäre Form des Dateischutzes, die aber verbessert werden muß. Es ist abzusehen, daß aufgrund immer leistungsfähigerer Hardware künftig auch Dateien auf Klientensystemen netzwerkweit verfügbar sein werden. Folglich wird ein ausgereifter Schutzmechanismus benötigt, der die Dateien auf Klientensystemen berücksichtigt. Ein Dateischutzmechanismus kann aber auch auf einem System für einen Benutzer nützlich sein, auf das nicht über ein Netzwerk zugegriffen wird. Der Hintergrund: Es gibt eine Vielzahl "trojanischer" Programme, die eine andere Aufgabe ausführen, als sie vorgeben auszuführen. In einer nicht vernetzten Umgebung stellen diese Programme lediglich Beispiele von Vandalismus dar; üblicherweise wird der Inhalt einer Festplatte oder einer Diskette gelöscht. In einer künftigen Büroautomationsumgebung können aber Lohn- und andere sensitive Dateien verändert werden (auch über das Netzwerk hinweg). Wenn der Benutzer sensitive Dateien unter dem Schutz eines Passwortes speichert, sind diese auch vor einem irrtümlich selbst gestarteten "trojanischen" Programm sicher. Das Programm kennt das Passwort nicht und der Benutzer wird es sicherlich nicht bereitstellen. Dieser Selbstschutz verhindert, daß unbekannte Programme die Datei zerstören.

Schutzmechanismen besitzen üblicherweise zwei Formen: *Zugriffslisten* und *Capability Tokens*. Ein Capability Token erlaubt den Zugriff auf ein Objekt, wenn das anfordende Programm (oder dessen Benutzer) das entsprechende Token bereitstellt, das in einer Vielzahl von Formen vorliegen kann. Passwörter können für Dateien oder Verzeichnisse vergeben werden (zum Beispiel in verfügbaren Produkten wie MS-Net und PC-Net). Dies sind Formen des Capability Tokens. Wenn das Passwort bereitgestellt werden kann, ist der Zugriff auf die Datei erlaubt. Beachten Sie, daß das Passwort mit einem Objekt und nicht mit dem Benutzer

verknüpft wird. Jeder Benutzer, der das Token kennt, hat auch Zugriff auf die Datei.

Eine Hauseingangstür ist ein gutes Beispiel für ein Capability Token. Der Zugang zum Haus hängt davon ab, ob Sie den Schlüssel - den Capability Token - besitzen. Er ist nicht davon abhängig, wer Sie sind. Falls Sie den Schlüssel nicht besitzen, können Sie nicht in das Haus gelangen (selbst wenn es Ihr eigenes Haus ist). Andererseits kann jeder, der den Schlüssel besitzt (gleichgültig wer) in das Haus gelangen. Die Benutzung eines Schlüssels hat Vorteile: Er kann verliehen und zurückgefordert werden. Capability Tokens sind flexibel, da sie weitergegeben werden können und nicht vom Eigentümer des Objektes abhängig sind.

Dieser Vorteil stellt aber auch gleichzeitig eine Gefahr des Capability-Token-Systems dar: Die Möglichkeit der Verteilung des Schlüssels besitzt den Nachteil, daß Duplikate des Schlüssels angefertigt werden können. Nachdem Sie den Schlüssel einmal herausgegeben haben, können Sie niemals sicher sein, ob Sie wieder über alle Schüssel und Duplikate verfügen. Zudem können Sie nicht festlegen, wer mit dem Schlüssel Zugriff auf Ihr Haus hat, und Sie können den Zugang zu Ihrem Haus nicht mehr zurückziehen, nachdem der Schlüssel einmal vergeben ist. Die einzige Möglichkeit, die vollständige Kontrolle über die Eingangstür des Hauses wiederzugewinnen, ist der Austausch des Schlosses (was aber bedeutet, daß der neue Schlüssel an jeden verteilt werden muß, der den Zugriff weiterhin behalten soll). Für Häuser und Schlüssel ist dies kein Problem, da Schlüssel nur selten verliehen werden. Die Veränderung eines Capability Tokens für eine Computerdatei ist schwieriger, da dies die Aktualisierung vieler Programme (die Zugriff darauf besitzen sollen) bedeutet. Wichtig ist dabei, daß die Aktualisierung aller Programme zu einem genau festgelegten Zeitpunkt erfolgt. Die Verteilung des neuen Capability Tokens muß mit Bedacht erfolgen, da eine falsche Verteilung wiederum zu einer Token-Änderung führen muß.

Hinzu kommt, daß ein Capability Token oder Passwort für jede Datei und jedes Verzeichnis gespeichert werden muß. Das bedeutet, daß das Passwort in Dateien, Batch-Scripts und an anderen Stellen des Systems abgelegt ist. Niemand kann sicherstellen, daß ein unbefugter Benutzer nicht eine Kopie des Passwortes erhält.

Eine Alternative ist der Mechanismus der *Zugriffslisten*. Dies entspricht der Überwachung der Personen und Objekte, die Zugriff auf ein Objekt besitzen. Jedes Objekt wird mit einer Liste verknüpft, in der aufgeführt ist, wer welche Zugriffsrechte besitzt. Zur Überprüfung, wer Zugriff besitzt, muß lediglich die Liste angezeigt werden. Durch einfaches Editieren der Liste können Zugriffsrechte gewährt oder gestrichen werden. Die Wartung ist einfach, da Veränderungen nur vorgenommen werden müssen, wenn Benutzer hinzugefügt oder aus der Liste entfernt werden sollen. Auch können in der Liste Gruppennamen (wie "jeder Mitarbeiter der Produktabteilung" oder "alle Vorstandsmitglieder") angegeben werden.

Das Problem dieser Methode liegt darin, daß das System Sie als Person erkennen muß. Der Computer verwendet dazu ein persönliches Passwort. Das Passwort zeigt nicht an, daß Sie Zugriff auf eine bestimmte Datei haben, es zeigt nur, daß Sie die Person sind, für die Sie sich ausgeben. Da Sie nur ein Passwort besitzen, können Sie dieses im Gedächtnis behalten; es braucht nicht in jedes Programm eingetragen werden. Das Passwort einer Person kann häufig geändert werden, wobei nur die die Veränderung ausführende Person davon Kenntnis hat. Nachdem dem Computersystem einmal bekannt ist, daß Sie tatsächlich die Person sind, für die Sie sich ausgeben, wird der Zugriff auf der Basis von Zugriffslisten erlaubt oder verweigert. MS-DOS kann dies nicht nutzen, da es ein ungeschütztes System ist, eine Flagge im Speicherbereich kann beliebig gesetzt werden. Auf diese Weise kann niemals sichergestellt werden, daß Sie der autorisierte Benutzer sind. OS/2 ist ein Betriebssystem, das den Protect-Modus verwendet und vor

derartigen Manipulationen sicher ist (vorausgesetzt, es werden keine Real-Modus-Anwendungen ausgeführt). Eine Netzwerkumgebung stellt eine weitere Herausforderung dar, da hier Programme entwickelt werden können (auf einem MS-DOS-System ablaufend, die den Schutzmechanismus umgehen), die das Netzwerk untersuchen und die Kommunikation überprüfen. Ein Klientensystem kann zur Identifikation eines Benutzers kein Passwort in das Netzwerk einspeisen, da dieses "abgehört" werden könnte. Auch kann nicht einfach eine Meldung mit dem Inhalt *Ich bin sicher, daß dies die autorisierte Person ist* gesendet werden, da auf dem Klientensystem fehlerhafte Software ablaufen könnte, die eine Erlaubnis ohne Berechtigung erteilt. Ein Netzwerkprotokoll muß annehmen, daß Programme alle Netzübertragungen abfangen, lesen und entsprechende Meldungen einspeisen können.

Es ist klar, daß das künftige Dateiverwaltungssystem von OS/2 objektorientierte Zugriffs- und Erlaubnislisten unterstützen wird. OS/2 wird dahingehend erweitert, daß Benutzer sich durch persönliche Passwörter identifizieren. Zukünftige Netzwerksoftware wird ein sicheres Netzwerkautorisierungsprotokoll unterstützen.

Das neue Dateiverwaltungssystem wird mehr Merkmale als nur die Unterstützung von Zugriffslisten mit sich bringen. So werden auch Dateinamen außerhalb der FAT-8.3-Konvention und erweiterte Dateiattribute unterstützt werden. Das heutige FAT-Dateisystem unterstützt nur wenige Attribute, die als binäre Flaggen implementiert sind - *System*, *verborgen*, *Nur-Lesen* usw. Erweiterte Attribute erlauben eine willkürliche Zusammenstellung von Attributen, die als Strings dargestellt und mit jeder Datei assoziiert werden. Einzelne Anwendungen werden die Definition bestimmter Attribute veranlassen können, diese lassen sich Dateien zuordnen und zu einem späteren Zeitpunkt abfragen. Erweiterte Attribute können beispielsweise zur Benennung der Anwendung eingesetzt werden, von der eine Datei angelegt wurde. Dies würde dem Benutzer das Anklicken mit der Maus (Mauszeiger befindet sich auf dem Dateinamen oder

Verzeichnis) erlauben, der Presentation Manager könnte die gewünschte Anwendung mit der Datei laden.

Obwohl ein solches Dateiverwaltungssystem einen guten Eindruck macht, kann nicht sichergestellt sein, daß alle zukünftigen Möglichkeiten und Erweiterungen unterstützt werden. Was ist mit neuen Datenträgern wie CD-ROM- und WORM-Laufwerken? Künftige OS/2-Versionen werden *installierbare Dateiverwaltungssysteme* unterstützen. Ein installierbares Dateiverwaltungssystem ähnelt den bekannten installierbaren Einheitentreibern. Bei der Initialisierung des Systems werden nicht nur die Einheitentreiber, sondern auch die Dateiverwaltungspakete von OS/2 installiert. So können Dateisysteme für heute noch ungewöhnliche Speichermedien wie CD-ROM- und WORM-Laufwerke eingesetzt werden und es wird eine Schnittstelle zu Dateisystemen außerhalb der DOS/OS/2-Welt geben.

Hinweis: CD-ROM-Laufwerke sind besondere Versionen der CD-Platten, auf denen Daten statt digitalisierter Musik gespeichert wird. Eine solche Platte besitzt ungefähr 600 Mbyte Nur-Lese-Speicherkapazität.

21.2 Der 80386

Durch das ganze Buch hindurch zieht sich der Begriff 80386 als eine Art Synonym für Höchstleistung hin. Nun, für den Systementwickler ist der 80386 ein Höchstleistungsprozessor. Er bietet die Schutzmechanismen des 80286, verfügt aber über drei weitere Schlüsselelemente.

21.2.1 Large-Segmente

Der 80386 besitzt eine segmentierte Architektur, die der des 80286 ähnelt. Die 80286-Segmente sind allerdings auf 64 Kbyte Speicherkapazität begrenzt. Für den 80386 können Segmente bis zu vier Millionen Kbyte groß sein. Es ist möglich, daß ein Programm in zwei Segmenten abläuft (eines für den Programmcode und eines für Daten). Die Segmentationsleistungsmerkmale des Prozessors können dabei außer acht gelassen werden. Dies wird *Flat Model* genannt. Die Entwicklung von Programmen, die mit umfangreichen Strukturen umgehen, ist unter Verwendung des

Flat Model einfacher, da Compiler bei der Generierung optimal segmentierten Programmcodes meist Schwierigkeiten haben. Die Umwandlung von 8086-/80286-Large-Model-Programmen in ein 80386-Flat-Model-Programm kann die Ausführungsgeschwindigkeit drastisch erhöhen. Obwohl künftige Versionen von OS/2 sicherlich Segmente und Anwendungen zulassen werden, die das Flat Model intern nutzen, muß OS/2 nicht unbedingt eine Flat-Model-API bereitstellen. Die System-API für 32-bit-Anwendungen kann weiterhin segmentierte Adressen (das bedeutet 48-bit-Adressen) einsetzen.

21.2.2. Mehrere Real-Modus-Boxen

Der 80386 unterstützt einen Ausführungsmodus, der *virtueller Real-Modus* genannt wird. Prozesse, die in diesem Modus ausgeführt werden, verhalten sich wie bei der Ausführung im tatsächlichen Real-Modus. Sie befinden sich in einem besonderen 8086-kompatiblen Protected-Modus. Die erweiterten Speicherverwaltungs- und Schutzleistungsmerkmale dieses Modus erlauben einer künftigen OS/2-Version die Unterstützung von mehr als einer Real-Modus-Box zu einem Zeitpunkt (auch im Hintergrund). Der virtuelle Real-Modus eliminiert die Notwendigkeit zur Modusumschaltung. Auf diese Weise kann der Benutzer Real-Modusanwendungen ausführen lassen, während gleichzeitig Kommunikationsanwendungen mit einer hohen Interrupt-Rate ablaufen.

21.2.3 Vollständiger Schutz

Der virtuelle Real-Modus mit dem 80386-Leistungsmerkmal zur Ermöglichung (oder Unterdrückung) des Ein-/Ausgabezugriffs auf Port-Basis stellt die Hardware-Grundlage für eine vollständig sichere OS/2-Version der Zukunft dar. Bei einem solchen OS/2 werden beim Start Module geladen (das Betriebssystem, Einheitentreiber, installierbare Dateisysteme und ähnliches), die vertrauenswürdig sein müssen. Keine anderen Programme können gewollt oder ungewollt Dateien zerstören, geschützte Dateien lesen oder Zugriff auf geschützte Daten erlangen. Die einzige Möglichkeit der Beeinflussung durch ein fehlerhaftes Pro-

gramm ist die Verringerung der Ausführungsgeschwindigkeit durch das Belegen von Ressourcen (wie dem umfangreichen Gebrauch von RAM oder von CPU-Zeit). Dies ist allerdings relativ harmlos, da der Benutzer das störende Programm einfach beenden kann.

21.2.4 Andere Leistungsmerkmale

Der 80386 verfügt neben der Verarbeitungsgeschwindigkeit über eine Reihe weiterer gravierender Leistungsmerkmale, die nicht in einer API erscheinen. Ein Beispiel stellt die seitenorientierte virtuelle Speicherverwaltung dar.

Was bedeuten die 80386-Leistungsmerkmale für den 80286? Welche Rolle wird der Prozessor in der Zukunft spielen? Soll ein Entwickler seine Programme für den 80286 oder für den 80386 schreiben? Zunächst einmal muß gesagt werden, daß OS/2 für den 80386 das gleiche Betriebssystem (in den wichtigsten Aspekten) wie für den 80286 ist. Die einzige neue API im 80386-OS/2 ist eine 32-bit-API, die zur Ausführung von Nur-80386-Binärprogrammen eingesetzt wird. Die anderen Leistungsmerkmale - wie virtueller Speicher, Ein-/Ausgabe, Zugriffserlaubnisüberprüfung und mehrere Real-Modus-Boxen sind für den Benutzer zwar wertvoll, machen aber keine neuen APIs erforderlich und sind daher zu allen Anwendungen kompatibel. Die Nutzung der neuen Anweisungen des 80386 oder der 2^{32}-byte-Segmente wird sicherlich eine neue API erfordern. Immerhin muß ein Programm für eine solche Umgebung speziell geschrieben und kompiliert werden. Nur Programme, die mit den 80286-kompatiblen Segmenten nicht funktionieren, sind 80386-abhängig. 80286-Programme, die im Protect-Modus ablaufen, können ohne Veränderung und Nachteile auf den 80386 übernommen werden. Bei diesen Programmen wird auf dem 80386 lediglich ein Geschwindigkeitsvorteil erkannt.

Zusammenfassend kann gesagt werden, daß es nur ein Betriebssystem OS/2 geben wird. OS/2 unterstützt 16-bit-Anwendungen im Protect-Modus, die auf allen Systemen ablaufen werden, und 32-bit-Anwendungen im Protect-Modus, die nur auf 80386-

Rechnern lauffähig sind. Ein Entwickler sollte die Programmierung einer Anwendung im 32-bit-Modell nur dann erwägen, wenn die Anwendung im 16-bit-Modell so wenig Durchsatz aufweist, daß das Programm quasi wertlos ist. Andererseits werden 16-bit-Anwendungen mit Sicherheit auf allen existierenden OS/2-kompatiblen Rechnern ablauffähig sein. Gleiches gilt für alle OS/2-Versionen. Nachdem der 80386 und OS/2-386 eine ausreichende Marktdurchdringung erfahren haben, kann die Umstellung und Neuentwicklung von Produkten auf der Basis des 80386 in Erwägung gezogen werden.

21.3 Die nächsten zehn Jahre

Microsoft glaubt, daß OS/2 in den nächsten zehn Jahren einen wesentlichen Einfluß auf die Personal-Computer-Branche haben wird. Die Standardisierung von Computerumgebungen, die die Massenvermarktung von Software mit sich bringt, gibt solchen Standards Langlebigkeit, während die rasche Verbesserung der Hardware einen großen Druck zum Wandel bringt. Als Resultat erwarten wir, daß OS/2 sehr lange aktuell bleiben wird, wobei "lang" sicher relativ ist. Wie kann ein Nachfolgesystem von OS/2 aussehen? Wenn diese Frage heute beantwortet werden könnte, wäre ein Nachfolgesystem nicht mehr nötig. Sicherlich wird die CPU-Leistung ständig wachsen. Personal Computer werden unzweifelhaft den Fußstapfen ihrer "Supercomputer-Brüder" folgen und künftig auch für anderes als für bloße Berechnungen (beispielsweise zur Simulation und als Expertensysteme) genutzt werden. Der Personal Computer wird nach der Arbeitsstätte auch die Privatsphäre der Menschen "erobern". Die Zukunft wird uns klarer erscheinen, wenn dieses Werkzeug beginnt, den Benutzer selbst zu verändern. Die Entwicklung von OS/2 stellt das größte Projekt dar, mit dem sich Microsoft jemals beschäftigt hat. Von einer anfänglich sehr kleinen Gruppe von Microsoft-Ingenieuren über ein immer noch recht kleines Microsoft-IBM-Jointventure-Team bis zu einer großen Anzahl von Entwicklern, Testern und Dokumentationsmitarbeitern sowohl von Microsoft als auch von IBM wurde das Projekt als "schwarzes Loch" bekannt. Während das Buch verfaßt wird, befindet sich

OS/2 in der Endphase, einige Wochen vor der allgemeinen Verfügbarkeit. Es war mir und meinen Mitarbeitern eine große Freude, zu erkennen, daß das "schwarze Loch" die Früchte unserer Arbeit nun endlich an die Benutzer zurückgibt.

Glossar

3X-Box

Die OS/2-Umgebung, in der ein PC auf der Basis des 80286 unter MS-DOS Versionen 2.X oder 3.X emuliert wird.

Allgemeine Prioritätskategorie

Die allgemeine Prioritätskategorie ist die OS/2-Klassifikation von Threads, die aus drei Kategorien besteht: Hintergrund, Vordergrund und interaktiv.

Allgemeiner Schutzfehler (GP)

Siehe *GP-Fehler*.

Anonyme Pipe

Ein von OS/2 im RAM gewarteter Datenpuffer, der zur Kommunikation zwischen Prozessen eingesetzt wird.

Applications Program Interface (API)

Eine Zusammenstellung von Programmaufrufen, die ein Programm zur Erlangung von Serviceleistungen des Betriebssystems verwendet. Der Ausdruck API steht für eine Serviceschnittstelle, gleich welcher Form.

Befehlsunterbaum

Ein Prozeß und alle Nachfolger.

Benannte Pipe

Ein von OS/2 im RAM gewarteter Datenpuffer, der zur Kommunikation zwischen Prozessen eingesetzt wird.

Bildschirmgruppe

Eine Gruppe von einem oder mehreren Prozessen, die (im allgemeinen in serieller Weise) einen einzigen logischen Bildschirm gemeinsam verwenden.

Call Gate

Ein besonderer LDT- oder GDT-Eintrag, der einen Unterroutineneinsprungspunkt und ein Speichersegment beschreibt. Ein *far*-Aufruf an einen *Call Gate*-Selektor führt zu einem spezifizierten Einsprungspunkt. Dies ist ein Leistungsmerkmal der 80286/80386-Hardware und wird normalerweise bei der Änderung eines niedrigen Privilegstatus in einen höheren Privilegstatus verwendet.

Daemonprogramm

Ein Prozeß, der eine Utility-Funktion ohne Interaktion mit dem Benutzer ausführt. Die Auslagerung eines Speicherteils ist beispielsweise ein Daemonprogramm.

Dateinamensbereich

Im Dateinamensbereich werden Namen gespeichert, die das Format von Dateinamen besitzen. In den ersten Versionen von OS/2 werden einige der Namen in internen RAM-Tabellen gehalten und existieren nicht auf einem Datenträger.

Dateinummer

Ein binärer Wert, der eine geöffnete Datei repräsentiert; die Dateinummer wird in allen dateiorientierten Ein- und Ausgabeaufrufen verwendet.

Dateisperre

Das OS/2-Leistungsmerkmal, das einem Programm die temporäre Sperrung einer Datei (Schutz vor Lesen und/oder Schreiben) erlaubt.

Datensatzsperre

Der Mechanismus, der einem Prozeß die Sperrung bestimmter Bereiche von Bytes in einer Datei erlaubt. Bei aktivierter Sperre kann kein anderer Prozeß diese Bytes lesen oder schreiben.

Debugger

Ein Programm, das dem Programmierer bei der Suche von Laufzeitfehlern beim Austesten eines Programms behilflich ist.

Durchschreiben

Eine Option beim Schreiben von Dateien. Das *Durchschreiben* führt dazu, daß die normale Zwischenspeicherung umgangen wird und die Daten direkt auf Diskette geschrieben werden.

Dynamische Verknüpfung

Dies ist die Auflösung externer Referenzen, die beim Laden oder beim Ablauf eines Programms stattfindet. Eine dynamische Verknüpfung erlaubt die Zusammenfassung der aufgerufenen Routinen in Pakete, die getrennt vom aufrufenden Programm verkauft und aktualisiert werden können. OS/2 setzt den Mechanismus der dynamischen Verknüpfung als primäres Systemmerkmal ein, über das Systemservices (und auch Services, die nicht vom System stammen) aufgerufen werden können.

Dynamische Verknüpfung beim Laden

Dies ist die Verknüpfung eines Klientenprozesses mit den Dynlink-Bibliotheken beim ersten Laden eines Prozesses in den Speicher.

Dynamische Verknüpfung bei der Ausführung

Dies ist die dynamische Verknüpfung, die erfolgt, nachdem die Ausführung eines Prozesses begonnen hat. An OS/2 werden dazu das Modul und die Einsprungspunktnamen übergeben; das Betriebssystem liefert die Adresse der Routine an aufrufende Programme.

Dynlink-Bibliothek

Eine Dynlink-Bibliothek ist eine Datei in einem speziellen Format, das den Programmcode einer Gruppe von dynamisch einzubindenden Unterroutinen in binärer Form enthält.

Dynlink-Routine

Siehe *dynamische Verknüpfung*.

Dynlink-Untersystem

Ein Dynlink-Modul, das Services bereitstellt, die sich auf eine Ressource beziehen.

Ein-/Ausgabeprivilegmechanismus

Ein Leistungsmerkmal, das Prozessen die Anforderung des direkten Zugriffs auf Ein-/Ausgabeports einer Einheit über den Einheitentreiber erlaubt. Gleiches gilt für dedizierte Speicherstellen sowie Memory-Mapped-Speicherstellen. Der Ein-/Ausgabeprivilegmechanismus kann entweder direkt von einer Anwendung oder indirekt über ein Dynlink-Paket verwendet werden.

Einheitentreiber

Ein Programm, das Ein-/Ausgabeanforderungen in einer standardisierten, einheitenunabhängigen Weise ausführt. Die Anforderungen werden in eine Form umgewandelt, die der anzusprechenden Hardware gerecht wird.

Einheitenüberwachung

Ein Mechanismus, der Prozessen die Überwachung und /oder Modifizierung von Einheitendatenströmen erlaubt.

Erzwungenes Ereignis

Ein Ereignis oder eine Handlung, die einem Thread oder Prozeß von einer externen Quelle aufgezwungen wird. Die Betätigung der Tastenkombination Ctrl-C oder ein DOSKILL-Aufruf ist ein solches Ereignis.

Exit-Liste

Eine Liste von Unterroutinen, die OS/2 nach der Beendigung eines Prozesses aufruft. Die Exit-Liste wird nach der Prozeßbeendigung, aber noch vor der Zerstörung des Prozesses ausgeführt.

Family Application Program Interface (Family API)

Eine Standardausführungsumgebung unter MS-DOS Version 2.X und 3.X und unter OS/2. Der Programmierer kann eine Family API für ein Anwendungsprogramm benutzen, die eine Untermenge der OS/2-Funktionen (umfangreicher als die unter MS-DOS 3.X vorliegenden Funktionen sind) einsetzt und in einer binär kompatiblen Weise unter MS-DOS Version 2.X und 3.X sowie unter OS/2 ablauffähig ist.

Filtermodus

Ein Modus, der von Programmen für die Tastatureingabe gesetzt werden kann. Im Filtermodus interpretiert OS/2 Steuerzeichen (wie *Rückschritt*) und überläßt die Bearbeitung nicht dem empfangenden Prozeß.

GDT

Siehe *globale Deskriptortabelle*.

Gemeinsam verwendeter benannter Speicher

Ein Speichersegment, auf das mehrere Prozesse gleichzeitig zugreifen können. Der Zugriff auf das Segment kann von Prozessen angefordert werden, wenn der Name des Segmentes bekannt ist.

Gemeinsam verwendeter Speicher

Ein Speichersegment, das gleichzeitig von mehreren Prozessen benutzt werden kann.

Geschlossenes System

Hardware oder Software, die von Drittanbietern nicht erweitert werden kann.

Globale Deskriptortabelle (GDT)

Ein Element der 80286/80386-Speicherverwaltungshardware. Die GDT enthält die Beschreibung von bis zu 4095 globalen Segmenten. Ein globales Segment ist allen Prozessen verfügbar.

Globale Untersysteminitialisierung

Ein Leistungsmerkmal, das einer Dynlink-Routine die Verwendung ihres Initialisierungseinsprungspunktes beim Aufruf des Dynlink-Paketes vom ersten Klientenprozeß erlaubt.

Globales Datensegment

Ein Datensegment, das von allen Instanzen einer Dynlink-Routine gemeinsam verwendet wird. In anderen Worten: Ein Segment, das allen Prozessen, die eine bestimmte Dynlink-Routine aufrufen, verfügbar ist.

GP-Fehler

Der Fehler tritt auf, wenn ein Programm auf ungültige Speicherstellen oder in einer nicht gültigen Weise auf gültige Speicherstellen zugreift (zum Beispiel ein Schreibvorgang in Nur-Lese-Speicherbereiche).

Großvaterprozeß

Dies ist der Vaterprozeß eines Prozesses, der wiederum einen Kindprozeß angelegt hat.

Hardwarefehler-Daemon

Ein Daemonprozeß, der Hardwarefehler bearbeitet. Der Hardwarefehler-Daemon kann ein unabhängiger Prozeß oder Thread sein, der zum Session Manager oder Presentation Manager gehört.

Hardwarefehler

Ein Fehler, der vom System festgestellt wird, ohne Benutzerinteraktion aber nicht korrigiert werden kann.

Hintergrundkategorie

Eine Klassifikation der Prozesse, die mit der Bildschirmgruppe verknüpft sind, die derzeit nicht auf dem Bildschirm angezeigt wird.

Huge-Segmente

Eine Softwaretechnik, die das Anlegen und die Verwendung von Pseudosegmenten erlaubt, die größer als 64 Kbytes sind.

Installierbares Dateiverwaltungssystem (IFS)

Ein Programmteil, das OS/2 beim Startvorgang lädt. Das Programmteil stellt die Software zur Verwaltung eines Dateisystems und Datenträgers einschließlich der Leistungsmerkmale zum

Anlegen von Verzeichnissen, Belegen der Speicherkapazität und anderer Leistungsmerkmale bereit.

Instanzdatensegment

Ein Speichersegment, das Daten für eine Instanz einer Dynlink-Routine enthält.

Instanzuntersysteminitialisierung

Ein Service, den Dynlink-Routinen anfordern können. Der Initialisierungseinsprungspunkt einer Dynlink-Routine wird jedesmal aufgerufen, wenn ein neuer Klient mit der Routine verbunden wird.

Interaktive Kategorie

Eine Prozeßklassifikation, die die momentan mit der Tastatur arbeitenden Prozesse umfaßt.

Interaktive Programme

Das Programm, das von der Tastatur oder aus einer anderen Quelle Daten annimmt (beispielsweise eine Textverarbeitung oder ein Tabellenkalkulationsprogramm). Programme wie Compiler können eingeschränkt interaktiv sein, indem sie nach Dateinamen und Kompilierungsoptionen fragen. Im allgemeinen werden sie aber nicht als interaktiv angesehen, da die Programmfunktion die Kompilierung des Quellcodes und nicht die Bereitstellung von Antworten auf Benutzeranfragen ist.

IPC

Siehe *Kommunikation zwischen Prozessen.*

Isolierung (Vereinzelung)

Das Prinzip des Verbergens der internen Implementierung eines Programmes, einer Funktion oder eines Services, so daß der Klient zwar die Aufgabe kennt, nicht aber die Art und Weise, in der die Aufgabe ausgeführt wird.

Kategorie niedriger Priorität

Eine Prozeßklassifikation, in der alle Prozesse enthalten sind, die nur dann CPU-Zeit erhalten, wenn kein weiterer Thread in anderen Kategorien diese benötigt. Die Kategorie ist in ihrem Prioritätsstatus niedriger als die Kategorie allgemeiner Priorität.

KBD

Eine Abkürzung für ein Dynlink-Paket, das die Einheit *Tastatur* behandelt. Die Bezeichnungen aller Einsprungspunkte beginnen mit *Kbd* (für *Keyboard*).

Kernel

Der Kernel ist der zentrale Teil von OS/2 und liegt permanent in vorgegebenen Speicherpositionen vor; er wird im Ring-0-Status (privilegiert) ausgeführt.

Kindprozeß

Ein Prozeß, der von einem anderen Prozeß (dem Vaterprozeß) angelegt wurde.

Kommunikation zwischen Prozessen (IPC)

Dies ist das Leistungsmerkmal eines Prozesses oder Threads zur Übertragung von Daten oder Meldungen zu einem anderen Prozeß oder Thread. Auf diese Weise können Dienste systemweit angeboten oder angefordert werden.

Kontextumschaltung

Die Umschaltung der CPU von der Ausführung eines Threads zu einem anderen. Der zweite Thread kann zum gleichen oder einem anderen Prozeß gehören.

Kritischer Bereich

Ein Programmteil, das Datenressourcen in einer eintrittsinvarianten Art verändert.

LDT

Siehe *lokale Deskriptortabelle.*

Logische Einheit

Ein symbolischer Name für eine Einheit, den ein Benutzer jeder physikalischen Einheit zuweisen kann.

Logisches Verzeichnis

Ein symbolischer Name für ein Verzeichnis, das ein Benutzer einem Laufwerk oder Verzeichnis zuweisen kann.

Lokale Deskriptortabelle (LDT)

Ein Element der 80286/80386-Speicherverwaltungshardware. Die LDT enthält die Beschreibung von bis zu 4095 lokalen Segmenten. Jeder Prozeß besitzt eine eigene LDT und kann nicht auf die LDTs anderer Prozesse zugreifen.

MOU

Dies ist eine Abkürzung für ein Dynlink-Paket, das die Einheit *Maus* behandelt. Die Bezeichnungen aller Einsprungspunkte beginnen mit *Mou* (für *Mou*se).

Multitasking-Betriebssystem

Ein Betriebssystem, in dem zwei oder mehr Programme/Threads zum gleichen Zeitpunkt ablaufen können.

Nicht-Filtermodus

Ein Modus, der von Programmen zur Übernahme von Tastatureingaben gesetzt werden kann. Im Nicht-Filtermodus übergibt OS/2 dem aufrufenden Programm jedes Zeichen wie es eingegeben wurde. Das aufrufende Programm ist für die Behandlung von Steuerzeichen (wie *Rückschritt*) verantwortlich.

Nummer

Ein willkürlicher Integerwert, den OS/2 an einen Prozeß zurückgibt. Die Nummer kann in nachfolgenden Aufrufen verwendet werden.

Objektnamenspuffer

Bereich, in den von OS/2 eine Zeichenkette geliefert wird, wenn der *DosExitPgm*-Aufruf erfolglos beendet wird.

Offenes System

Hardware oder Software, die Drittanbietern eine Verbesserung oder Aktualisierung erlaubt. Der Aufbau eines offenen Systems muß immer klar erkennbar sein.

Paralleles Multitasking

Ein Vorgang, bei dem mehrere Programme tatsächlich gleichzeitig ausgeführt werden (im Gegensatz zum seriellen Multitasking).

Physikalischer Speicher

Hauptspeicher (RAM, *random access memory*), der tatsächlich im System enthalten ist (im Gegensatz zu virtuellem Speicher).

PID (process identification number)

Ein einzigartiger Code, den OS/2 einem Prozeß bei der Erstellung zuweist. Die PID kann jeden Wert außer 0 annehmen.

Pipe

Siehe *anonyme Pipe*.

Presentation Manager

Der Presentation Manager ist die graphische Benutzerschnittstelle von OS/2.

Priorität (CPU-Priorität)

Dies ist ein numerischer Wert, der jedem ausführbaren Thread im System zugeordnet ist, um Ablaufprioritäten für die Threads festzulegen.

Privilegierter Modus

Ein Ausführungsmodus (auch als Ring 0 bekannt), der von der 80286/80386-Hardware unterstützt wird. Ein in diesem Modus ausgeführter Programmcode kann bestimmte Anweisungen verwenden, um Systemstrukturen und -tabellen zu beeinflussen. Lediglich der OS/2-Kernel und Einheitentreiber werden in diesem Modus ausgeführt.

Protect-Modus

Der Ausführungsmodus des Mikroprozessors 80286, der dem Betriebssystem die Verwendung von Leistungsmerkmalen zum Schutz der Anwendungen gegeneinander erlaubt. Der Modus wird auch als *Protected*-Modus bezeichnet.

Prozeß

Ein Prozeß ist die Ausführungsinstanz einer binären Datei. Unter OS/2 sind die Ausdrücke *Task* und *Prozeß* austauschbar. Ein Prozeß ist eine Einheit, die Zugriffsberechtigungen auf Ressourcen besitzen kann. Prozesse können Ressourcen wie Speicherbereiche, geöffnete Dateien, Dynlink-Bibliotheken und Semaphoren besitzen.

RAM-Semaphore

Eine Semaphore, die so im Hauptspeicher vorliegt, daß ein Thread Zugriff darauf hat. Diese Art der Semaphoren ist schnell, besitzt aber nur eine begrenzte Funktionalität. Siehe *Systemsemaphoren*.

Real-Modus

Das ist der Ausführungsmodus des Mikroprozessors 80286, in dem Programme ablaufen, die für die Mikroprozessoren 8086 oder 8088 entwickelt wurden.

Reservierter LDT-Bereich

Im reservierten LDT-Bereich liegen die LDT-Selektoren vor, die der Beschreibung von gemeinsam genutzten Speicherobjekten dienen oder von mehreren Prozessen in gleicher Weise verwendet werden.

Ring 3

Ring 3 ist die Privilegebene, mit der Anwendungsprogramme ausgeführt werden. Auf dieser Ebene ausgeführter Programmcode kann Systemstrukturen nicht modifizieren.

Scheduler

Der Scheduler ist der Teil von OS/2, der entscheidet, welcher Thread für welche Zeitspanne ausgeführt wird. Durch den Scheduler werden auch die Prioritäten der Threads festgelegt.

Semaphore

Eine Semaphore ist eine Softwareflagge oder ein Signal, das zur Koordination der Ausführung von zwei oder mehr Threads verwendet wird. Üblicherweise wird damit ein kritischer Bereich geschützt.

Serielles Multitasking

Das serielle Multitasking ist ein Prozeß, bei dem mehrere Programme "scheinbar gleichzeitig" ausgeführt werden. Durch die sehr schnelle Umschaltung zwischen den einzelnen Programmen bemerkt der Benutzer nicht, daß jeweils nur ein Programm zu einem Zeitpunkt bearbeitet wird.

Session Manager

Der Session Manager ist eine System-Utility, die die Umschaltung zwischen Bildschirmgruppen verwaltet. Er wird nur dann eingesetzt, wenn kein Presentation Manager vorhanden ist; der Presentation Manager ersetzt den Session Manager.

Signale

Ein Signal ist ein Mechanismus, der in der Software implementiert ist und in gleicher Weise wie ein Hardwareinterrupt funktioniert.

Signalgabe

Die Signalgabe beschreibt die Verwendung von Semaphoren, um Threads mitzuteilen, daß bestimmte Ereignisse oder Handlungen stattgefunden haben.

Singletasking-System

Singletasking ist das Gegenteil von Multitasking. In einem Singletasking-System kann nur jeweils ein Task zu einem Zeitpunkt ablaufen.

Software Tools Approach

Software Tools Approach ist eine Design-Philosophie, bei der jedes Anwendungsprogramm in einem Programmpaket eine bestimmte Aufgabe übernimmt und diese möglichst optimal erfüllt.

Speicherauslagerung

Die Speicherauslagerung (*swapping*) ist eine Technik, bei der ein Teil des Programmcodes und/oder der Daten im Speicher auf einen externen Datenträger ausgelagert wird. Auf diese Weise kann mehr Hauptspeicher vergeben werden als physikalisch im System vorhanden ist.

Speicherunterbelegung

Die Speicherunterbelegung ist das OS/2-Leistungsmerkmal zur Belegung von Teilstücken des Speichers aus einem Anwendungssegment heraus.

Speicherverwaltung

Die Speicherverwaltung belegt sowohl physikalischen als auch virtuellen Speicherbereich unter OS/2.

Stack-Frame

Der Stack-Frame ist ein Teil des Stapels eines Threads, der die lokalen Variablen und Parameter für eine Prozedur enthält.

Statische Verknüpfung

Die statische Verknüpfung ist das Ergebnis mehrerer Compilerläufe, die zu einer ausführbaren Datei gebunden wurden. Beim Binden (Linken) werden externe Referenzen aufgelöst.

Systemdateitabelle (SFT)

Die Systemdateitabelle ist eine interne OS/2-Tabelle, die einen Eintrag für jede geöffnete Datei enthält.

Systemsemaphore

Eine Systemsemaphore ist im internen Speicherbereich von OS/2 implementiert; sie arbeitet etwas langsamer als eine RAM-Semaphore, stellt dafür aber erweiterte Leistungsmerkmale zur Verfügung.

Task

Siehe *Prozeß*.

Thread of Execution

Dieser Thread wird gerade durch die CPU bearbeitet.

Thread-ID

Die Thread-ID ist die Kennummer eines Threads in einem Prozeß.

Thread

Ein Thread arbeitet einen Teil eines Anwendungsprogramms ab, wodurch mehr als ein Ausführungspfad für dieselbe Instanz eines Anwendungsprogramms möglich ist.

Umgebungsstring

Dies sind benutzer- oder programmiererdefinierbare Zeichenketten, die den einzelnen Prozessen zugeordnet sind. Die Initialisierung der Umgebungsstrings ist Aufgabe des Vaterprozesses.

Unterbrechbare Blockierung

Die unterbrechbare Blockierung ist eine besondere Form der blockierenden Operationen, die innerhalb des OS/2-Kernels verwendet werden. Damit kann ein Thread, der sich im OS/2-Kernel-Modus befindet und auf ein Ereignis wartet, aufgrund von äußeren Ereignissen (wie der Prozeßbeendigung oder der Betätigung der Tastenkombination Ctrl-C) vorzeitig zurückkehren.

Übergabespeicher

Ein Mechanismus zur Weitergabe gemeinsam verwendeter Speicherbereiche steht durch den Übergabespeicher bereit, bei dem ein Prozeß, der Zugriff auf ein Segment besitzt, die Zugriffserlaubnis an einen weiteren Prozeß übergeben kann. Prozesse können keine Zugriffserlaubnis anfordern; der Zugriff muß von einem Prozeß, der bereits Zugriffsrechte besitzt, weitergegeben werden.

Vaterprozeß

Prozeß, der einen anderen Prozeß (den Kindprozeß) anlegt.

Verborgener Thread

Thread, der von einem Dynlink-Paket angelegt wurde und im Dynlink-Programmcode verbleibt. Er kehrt niemals zum Programmcode des Klientenprozesses zurück; der Ausdruck wird auch für einen Thread verwendet, der einen Serviceeinsprungspunkt anspricht und vom Aufruf nicht zurückkehrt oder nur dann zurückkehrt, wenn ein bestimmtes Ereignis auftritt.

VIO

Dies ist eine Abkürzung für ein Dynlink-Paket, das die Einheit *Bildschirm* behandelt. Die Bezeichnungen aller Einsprungspunkte beginnen mit *Vio* (für *Vi*deo).

Virtualisierung

Dies ist die allgemeine Technik des Verbergens einer komplizierten Programmsituation hinter einer einfachen, standardisierten Schnittstelle.

Virtueller Speicher

Der Speicherbereich, der von einem Prozeß belegt und verwendet wird. Beim Zugriff auf den Speicherbereich muß sich der virtuelle Speicher im physikalischen Speicher befinden; zu jedem anderen Zeitpunkt kann er auf einen externen Datenträger ausgelagert sein.

Vordergrundkategorie

Eine Prozeßklassifikation; das sind die Prozesse, die in der gerade aktiven Bildschirmgruppe ablaufen.

Warteschlange

Eine Liste von Elementen, die auf die Ausführung warten.

Zeiteinheit

Die Ausführungszeit, die der Scheduler einem Thread gewährt, bevor die CPU-Rechenkapazität einem anderen Thread gleicher Priorität zugewiesen wird.

Zeitkritische Priorität

Eine Prozeßklassifikation, die interaktiv oder nicht-interaktiv sein kann. Sie kann in der Vorder- oder Hintergrundbildschirmgruppe vorliegen und besitzt eine höhere Priorität als jeder nicht-zeitkritische Thread im System.

Sachwortverzeichnis

& 238
.DDL, Datei 118, 127, 131, 137, 142, 148
.EXE, Datei 115, 321, 324
.OBJ, Datei 113
1-wait-state-Speicher 194
16-bit-Ganzzahlwert 58
16-bit-Mikroprozessor 5
24 INT 303
32-bit-API 338
32-bit-Datenträgerkennummer 256
32-bit-Zufallszahl 258
3X-Box 177, 181, 313, 341
-, Bildschirmgruppe 320
-, Emulation 317
4-byte-Datenstruktur 194
64-Kbyte-Segment 179
8-bit-Mikroprozessor 5
8.3, Format 149, 152, 335
80286 8, 17, 253, 306, 314, 320, 336
-, kompatibel, Segment 338
-, Umgebung 174
80386 14, 17, 93, 308, 336
-, Architektur 174
-, Binärprogramm 338
8080 5
8086 9
8088 5, 27

Abarbeitung
-, simultan 91
-, Warteschlange 212
Abbrechen 299
Abfangen, Tastatureingabe 42
Abgebrochen, Thread 96
Abgehängt 81, 133, 144, 240
Ablauf, störungsfrei 38
-, Geschwindigkeit 33, 55, 182
-, optimiert 168
-, Programm 271
Ableger, überlebend 76
Ablegerprozeß 70, 239
Abschalten, System 187
Absolut/Relativ-Mechanismus 44
Abspeicherung 92
Absturz, System 40, 46
Abtreten, Zugriffserlaubnis 239
Abwärtskompatibilität 27, 30, 155
Additionsbeispiel 72
Adresse
-, Bereich 312
-, Offset 124
-, signalbearbeitend 221
Adressierbar 34, 167
Adreßtabelle 124
Aktiv, Prozeß 133
Aktivierung, ausgelagertes Programm 175
Aktualisierung 253
-, Bildschirmdarstellung 15, 228, 235
-, gleichzeitig 94
-, Journaldatei 132
-, statisch, Daten 95
-, Version, Dritthersteller 37
Akzeptanz, Standard 251
Alarmsystem 93
All-Round-Paket 21
Allgemein
-, Priorität 101, 341
-, Schutzfehler 341
Alt-Esc 47
Analyse 3
Anforderung
-, Ein-/Ausgabeoperation 105
-, Semaphore 194
-, Verweigerung 27
Anfrage 163
Angepaßt
-, Priorität 103
-, Umgebung 12
Angezeigt, Eingabeaufforderung 248
Animation 12
Ankommend, Signal 311
Anonym, Pipe 64, 83, 341, 202
Antwort
-, Benutzer 302
-, schnellstmöglich 103
Antwortcode 302
Antwortfähigkeit 12
Antwortverhalten 180
Antwortzeit 102, 180
Anweisung
-, Call 122
-, Lade Segmentregister 178
Anwendung
-, graphisch, Unterstützung 12
-, interaktiv 242
-, Querverbindung 82
-, Real-Modus 25
-, schnell umschalten 16
-, zeitkritisch 163, 282
Anwendungsmodus 300

Anwendungsprogramm
-, direkter Zugriff 11
-, Hilfsmittel 25
-, Schnittstelle 26
-, störungsfreier Ablauf 38
-, umschalten 53
Anwendungssoftware 184
Anzeige, Meldung 303
Anzeigeadapter 13
Anzeigeeinheit, virtuell 13
API 31, 161, 205, 317, 329, 338, 341
-, applications program interface 26
-, Pipe, benennen 204
-, Schnittstelle 227
Application Program Interface 161, 194, 341
Applikation, PM 227
Applikationsstandard 26
Arbeit, mechanisieren 3
Arbeitsablauf, Unterbrechung 199
Arbeitshilfe 208
Arbeitsplatzcomputer 237
Arbeitsverzeichnis 43
Arbeitswarteschlange 93
Arbeitsweise, OS/2 62
Architektur
-, 80386 174
-, Einheitentreiber 14
-, Kernel 309
-, tool-basiert 193
-, von-Neumann 38
Argumentwort 216
ASCIIZ-String 36
Asynchron
-, Ein-/Ausgabe 253
-, Ereignis 93
-, Programmausführung 79
-, seriell, Port 93
-, Verarbeitung 92
Attribut 71, 335
Aufbau, modulares System 49, 55
Aufeinanderfolgend, Selektor 261
Aufgabe, finanzwirtschaftlich 48
Auflösung, Zeitgeber 280
Aufruf 231
-, DosCreateSem 195
-, DosExecPgm 79
-, far 121, 137, 158
-, indirekt 119
-, IOCTL 290
-, Namensbasis 260
-, Ring-0-Code 138
Aufrufreferenz 114
Aufrufsequenz 265
Aufrufverknüpfung 286
Aufsplittung 184
Aufwärtskompatibilität 30, 35
Auseinanderliegend, gleichweit, Selektor 167
Ausführbar, Family API 322
Ausführung
-, Adresse 323
-, Anwendungsprogramm 25
-, beschleunigt 93
-, Datensegment 168
-, Disketten-/Plattenoperation 103
-, Geschwindigkeit 19, 93
-, gleichzeitig 94
-, Programm 244
-, synchron 79
-, Synchronisation 80
-, Thread 219
Ausgabe
-, Architektur 237
-, asynchron 253
-, Eingabe, interaktiv 241
-, Fenster 109
-, Geschwindigkeit 257
-, Operation 105, 257
-, -, explizit 233
-, Port 308
-, Tastatur 248
Ausgelagert
-, interruptbearbeitender Code 177
-, Speicherbereich 175
Ausgereift, Schutzmechanismus 332
Auslagerung 175, 180
-, Algorithmus 177
-, Datei 19
-, Einheit 183
-, Kapazität 179, 180
-, Segment 99, 145, 314
-, Speicherbereich 17
-, Systemsegment 177
-, Verwaltung 177
-, vollständiger Prozeß 175
-, Vorgang 178
Austauschbar, Medium 257
Austesten 95
Ausüben, Nummernzugriff 244
Automatisch, Benennung 259
Availability 185

Basis
-, Leistungsmerkmal 34
-, Priorität 105, 108
-, Segment 165
-, Selektor 167
Bearbeiten, DosExitList-Routine 173
Bearbeitung
-, erzwungen 268
-, gleichzeitig 16

Bedieneroberfläche 227
Beeinflussung, gegenseitig 43
Beenden
-, Gesamtprozeß, Form 98
-, Thread 98
Beendet
-, String 36
-, vorzeitiger Prozeß 215
Beendigen
-, andere Programme 19
-, markierter Prozeß 88
-, Operation 292
-, Prozeß 98, 130
Beendigung
-, Kindprozeß 244
-, Kontrolle 132
Befehl
-, Anwendung 245
-, Eingabeschleife 248
-, Puffer 39
-, Unterbaum 75, 84, 87, 341
-, Unterbaumform 76
Begrenzungszeichen 149
Belegen, Speicher 137
Belegt
-, Speicherbereich 18
-, Thread, Speicherplatz 99
Belegung
-, Einheit 330
-, Fragmentierung 330
-, Kette, Datenbankverzeichnis 277
-, Systemressource 194
-, Tabelle 278
Benannt
-, nichtgeöffnete Pipe 208
-, Pipe 64, 154, 209, 211, 342
Benennen
-, API 204
-, gemeinsamer Speicher 161, 193
-, Pipe 194, 202, 203
-, Serviceanforderung 162
Benennung, automatisch 259
Benutzer
-, Anforderung 29
-, Antwort 302, 316
-, Beachtung 55
-, Eingabe 92, 178
-, Identifikation 335
-, Interaktion 20, 81, 91
-, Modus 13, 264
-, Schnittstelle 15, 109, 237
- ,-, graphisch 12, 223, 227, 330
-, Schnittstellenmodell 223
-, Schutz 47
Benutzt, gemeinsamer Speicher 193
Benutzung
-, exklusiv 197
-, gemeinsames Programm 118
Berechtigung 154
Bereich
-, Disjoint 124
-, kritisch 97, 174, 269, 317, 349
-, -, Schutz 196
-, -, Semaphore 198
-, -, Unterbelegungspaket 171
-, laden 176
-, LDT reservieren 125, 164
-, wiederherstellen 275
Bereit, Diskettenlaufwerk 47
Beschäftigt, Drucker 106
Beschleunigt, Ausführung 93
Betätigung, Umschalttaste 42
Betriebssystem 337
-, Architektur 138
-, Aufruf 97
-, Eingriff 30
-, Flexibilität 37
-, geschützt 38
-, graphikorientiert 109
-, interne Struktur 35
-, kompatibel 329
-, Multitasking 350
-, Singletasking 300
-, standardisierte Schnittstelle 26
-, Umgebung, Virtualisierung 43
-, Version 136
Beziehung, Eltern, Kind 239
Bibliothek
-, Dynlink 134, 324, 344
-, Routine 37, 115
Bildschirm
-, Abbildung, erlaubt 233
-, Anzeige, Aktualisierung 228
-, Controller 129, 229, 234
-, Darstellung 226
-, -, Aktualisierung 15
-, -, Hintergrund 235
-, -, hochauflösend 15
-, -, löschen 230
-, -, rollen 230
-, Daten 297
-, Einheit 71
-, Einheitentreiber 60, 297
-, Gruppe 46, 82, 102, 110, 228, 342
-, -, korrespondierend 246
-, -, Mechanismus 301
-, -, Programmcode 302
-, -, Prozeß 237
-, -, Real-Modus 110, 313
-, -, Umschaltung 301
-, -, virtuell 238
-, Inhalt, sichern 229

-, Karte 13
-, Kontrolle 233
-, orientiert, Schnittstelle 318
-, Speicher 229, 285
-, Speicherinhalt 229
-, sperren 230
-, Tastatur, Zugriff erben 243
-, Umschaltung 232, 302
-, Umschaltungssignal einfügen 267
-, Zugriff 239
Binärprogramm 31, 57, 318
-, 80386 338
Binden 113
BIOS
-, Interrupt 160
-, Druckvektor 263
Bit, nicht vererbbar 70
Blockieren 220, 273, 288
-, gegenseitig 208
-, Systemaufruf 103
-, unterbrechbar 310, 356
Blockorientiert
, Einheit 289, 291
-, Operation 230, 261
Break, Ctrl 88
Budgeanalyse 56
Bündelung, Funktionalität 137
Bürobetrieb, automatisiert 20
Byte
-, Kopie 260
-, orientiert 203
-, Strom 203

C 218
C-Bibliotheksroutine 68
Call Gate 137, 342
Call, Anweisung 122
CallNmPipe 208
Capability Token 332
CD-ROM 336
CGA 285
Chart 12
CHKDSK 276
Circuit, Virtual 29
CLI 172, 218, 297
Cluster 254
CMD.EXE 64, 82, 190, 238, 245
Code
-, Adresse 298
-, ausgelagert 177
-, einheitenspezifisch 13
-, extern, Datenbedarf 122
-, Kernel 96, 137, 177
-, Referenz 123
-, Segment 125, 168
-, -, Pure 142
-, Sequenz, Schutz 230
-, Umfang 33
CodeView 81
COMMAND.COM 314
Computerumgebung 339
CONFIG.SYS 180
Controller
-, Bildschirm 129
-, Laufwerk 293
Counting-Leistungsmerkmal 195
CP/M-80 5
CP/M 34
CPU 16, 54
-, Arbeitszeit 310
-, Ein-/Ausgabebindung 104
-, Geschwindigkeit 279
-, Priorität 202, 353
-, Priorität, verändern 87
-, Rechenzeit 100
-, Registerinhalt 90
-, Ressource 101
-, Umschaltung 317
-, Zeit 12, 44, 338
CreateThread 90
CS, Register 158
Ctrl
-,Break 79, 88, 217, 222, 245
-, C 74, 79, 171, 215, 220, 245, 311
-, Esc 47

Daemon
-, Hardwarefehler 301, 316, 347
-, Mechanismus 257
-, Programm 342
-, Prozeß 223
Datagram-ähnlich, Mechanismus 208
Datei
-, .DLL 118, 127, 131, 137, 142, 148
-, .EXE 115, 321. 324
-, .OBJ 113
-, Attribut, erweitert 335
-, Belegungstabelle 149
-, Durchsatz 330
-, Format 22, 151
-, Inhalt 44
-, Modifikation 44
-, Name 149, 152, 190
-, -, ungültig 47
-, -, zerlegen 36
-, Namenergänzung 149
-, Namensbereich 154, 342
-, Nummer 59, 131, 146, 277, 342
-, -, dupliziert 66
-, -, geöffnet 58
-, -, schließen 71
-, orientiert, Operation 260

-, öffnen 70, 137
-, Schutz 331
-, Sperre 28, 44, 303, 343
-, System 155, 166, 251, 330
-, -, installierbar 337
-, -, Namensbereich 155, 203
-, Transfermöglichkeit 109
-, Verwaltungssystem 7, 109, 149, 251, 278, 313, 331
-, -, installierbar 150, 336, 347
-, Verwendung 44
-, Zugriff 28
Daten 122
-, Adresse 213, 298
-, Aktualisierung, statisch 95
-, Austausch 22, 135, 214
-, Bank, Server 162
-, Bankverwaltungssystem 48, 277
-, Bedarf, extern 122
-, Bereich, statisch 122
-, Bereichsmechanismus 96
-, Durchsatz 223
-, Dynlink-Routine 122
-, Fehler 303
-, Fernübertragungsprogramm 107
-, Format, Kalkulationstabelle 135
-, global 127
-, Integrität 263, 269, 299
-, Kennsatz 255
-, Kennummer 256
-, -, Startsektor 260
-, löschen 276
-, Manipulation, routinemäßig 3
-, Menge, groß 15
-, Name 255, 258
-, Paket 106, 210
-, Puffer 64, 72
-, Referenz 123
-, Ressource 196
-, Risiko 141
-, Satz 116
-, Satzsperre 44, 273, 343
-, Schutz 20
-, Segment 91, 123, 168
-, -, Ausführung 168
-, -, Einheitentreiber 294
-, -, global 346
-, -, Selektor 140
-, sensitiv 20
-, Sicherheit, Dynlink 140
-, Strom, Zugriff, Einheit 264
-, Struktur 194, 269
-, Träger, prüfen 276
-, -, umfangreich 330
-, -, Wechsel 257
-, -, Treiberzugriff 177
-, Übertragung 4, 103, 163
-, Zwischenspeicherung 152
DDE 214
Deadlock 172, 195, 222, 233, 273
Deamon, Initiierung 134
Debugger 305, 308, 312, 343
-, Programm 308
-, Schnittstelle 309
-, Thread 311
Dediziert, Prozeß 134, 141
Definiert, Symbol 113
Deklaration, Fehler 311
Deskriptortabelle, lokal 350
Desktop-Publishing-System 4
DETACH 238
Detached 240
Detail
-, Implementierung 135
-, verbergen 216
DevDone 291
Developer's Kit 320
DevHlp
-, Adresse 285
-, Einsprungspunkt 285
-, Funktion 289
-, Schnittstelle 297
Device monitor 264
Diagnostizieren, Problem 19
Dichte
-, hoch 259
-, niedrig 259
Dienstleistungsroutine 67, 193
Digitalisierungstablett 23
Direkt
-, Kindprozeß 76, 85
-, Kommunikation 67
-, Umgebungskontrolle 313
-, Zugriff 11, 23, 242, 330
Disjoint, Bereich 124
DISKCOMP 260
DISKCOPY 260
Diskette 7
-, Ausführung 103
-, Ausgabe 220
-, Belegungstabelle, FAT 26
-, Datenübertragung 103
-, Eingabe 220
-, Kennummer 258
-, Laufwerk 47, 187, 289
-, MS-DOS 259
-, niedrige Dichte 259
-, Speicherplatz 254
-, umbenennen 259
-, Verwaltung 11
DLL 118
Dokumentationsmitarbeiter 339

DOS
-, Emulation 8
-, Leseaufruf 60
-, Schreibaufruf 60
-, Version 252
DosAllocHuge 164
DosAllocSeg 164, 321
DosAllocShrSeg 161
DosBufReset 260, 276
DosCall 137
DosCallNmPipe 208
DosClose 66, 151
DosConnectNmPipe 205
DosCreateCSAlias 142, 168
DosCreateSem 195
DosCreateThread 108
DosCWait 76, 83, 240
DosDevIOCtl 264, 306
DosDisconnectNmPipe 206
DosDupHandle 65, 73
DosEnterCritSec 97, 273
DosErrClass 136
DosError 147
DosExec, abgekoppelte Form 144
DosExecPgm 66, 77, 88, 108, 133, 178, 189, 243
DosExit 98, 132
DosExitCritSec 97
DosExitList 132, 144, 172, 200, 274, 310
-, bearbeitende Routine 173
DosFindFirst 325
DosFindNext 151
DosFlagProcess 216
DosFreeModule 120
DosFreeSeg 162, 214
DosGetAddr 119
DosGetHugeShift 164
DosGetMachineMode 325
DosGetProcAddr 119
DosGiveSeg 162
DosHoldSignal 172, 200, 218, 222, 275
DosKill 217, 248, 310
DosKillProcess 76, 88
DosLoadModule 119, 124, 144
DosMakeNmPipe 205
DosMakePipe 205
DosMonRead 267
DosMonReq 264
DosMonWrite 267
DosMuxSemWait 199, 254
DosNewSize 278
DosOpen 69, 73, 151, 190, 204, 260, 321
DosOpenSem 195
DosPeekNmPipe 210
DosPeekQueue 212
DosPtrace 81, 311
DosQHandType 242
DosRead 60, 65, 68, 90, 202, 208, 253, 261, 321
DosReadAsync 253
DosReadQueue 211
DosReallocHuge 166
DosResumeThread 97
DosScanEnv 191
DosSearchPath 191
DosSeek 65
DosSelfHandState 70
DosSemClear 196
DosSemRequest 99, 196, 201
DosSemSet 198, 275
DosSemWait 198, 202, 220
DosSetPrty 88, 108
DosSetSignalHandler 132, 218
DosSetVerify 147
DosSleep 92, 220, 274, 281
DosSubAlloc 169, 321
DosSubFree 169
DosSuspendThread 96
DosTimerAsync 281
DosTimerStart 281
DosTimerStop 281
DosTransactNmPipe 208
DosWaitNmPipe 210
DosWrite 65, 68, 90, 202, 205, 208, 253, 261, 276
DosWriteAsync 253
DosWriteQueue 211
DPATH 190
Dritthersteller, Aktualisierung 37
Drop-Down-Menü 223
Druck 55, 106
-, Ausgabe 268
-, Spooler 268
-, Vektor 263
-, Vorgang 106
-, Warteschlange 106
Dual-Modus 297
Duplikat 333
Dupliziert, Dateinummer 66
Durchsatz 99, 102, 176, 330
-, Verlust 182
Durchschreibedatei 278
Durchschreiben 276, 343
-, gesamter Prozeß 277
Dynamic Link 15, 113, 118
Dynamisch
-, Datenaustausch 214
-, Verknüpfung 15, 38, 91, 113, 116, 119, 224, 323, 343

Dynlink 264
-, Bibliothek 57, 134, 324, 344
-, Datensicherheit 140
-, Deamon-Initiierung 134
-, Einsprungspunkt 128
-, Modul, Nebeneffekt 146
-, Name 148
-, Paket 118
-, Plotter, Treiber 120
-, Routine 131, 219, 344
-, Subsystem 129, 163, 194
-, Symbol 128
-, Untersystem 344

Echt, Einheitentreiber 224
Echtzeitanwendung 279
Effizient
-, Ein-/Ausgabe 261
-, Gesamtsystem 305
-, Kommunikationsmöglichkeit 22
EGA 285
-, kompatible Hardware 231
Eigentümer, Objekt 333
Ein-/Ausgabe
-, Effizienz 261
-, Privilegmechanismus 305, 344
Einbenutzersystem 182
Einfügen,
  Bildschirmumschaltungssignal 267
Eingabe
-, Architektur 237
-, asynchron 253
-, Aufforderung 191
-, -, anzeigen 248
-, -, erneut 47
-, Geschwindigkeit 257
-, interaktiv 241
-, Korrektur 47
-, Operation 233, 257
-, Port 308
-, Tastatur 248
-, Zeile 73
Einheit
-, abhängig 229
-, Bezeichnung, logisch 189
-, blockorientiert 289
-, Datenstromzugriff 264
-, direkter Zugriff 23, 242
-, Diskettenspeicherplatz 254
-, interruptgesteuert 139, 287
-, logisch 188, 350
-, Monitor 266
-, Name 189
-, physikalisch 188
-, schließen 70
-, Schnittstelle 224
-, Speicher 306
-, Spezifisch, Code 13
-, Treiber 8, 139, 257, 283, 316, 337
-, -, blockorientiert 291
-, -, Datensegment 294
-, -, echt 224
-, -, fehlerlos 285
-, -, Programmierung 287
-, Treiberarchitektur, traditionell 23
-, Überwachung 264, 345
-, Verwaltung 297
Einheitlich, Schnittstelle 31
Einrichtung, IPC 134
Einsprung, Routine 294
Einsprungpunkt 116, 130, 265, 285
-, DevHlp 285
-, Dynlink 128
-, Name 148
Eintrag
-, History 76
-, LDT 123
Eintrittsvariant 196, 288
Einzelanwendung, spezialisiert 22
Einzeln, Prozeß 85
Elektronisch, Post 16
Element, Prozeß 89
Eltern, Kind, Beziehung 239
Empfänger, Selektor 213
Emulation 24, 317
End of file 72
Entdecken, Signal 220
Entscheidungsfindung 3
Entwicklung, Massensoftware 251
Entwurfsphilosophie 53
Environment Manager 243
EOF 72, 206, 240
Erbe 57
Erben, Zugriff 243
Ereignis 42
-, asynchron 93
-, erzwungen 310, 345
Erlaubnis
-, Bildschirmabbildung 233
-, Liste 335
Erneut
-, Eingabeaufforderung 47
-, Pipe öffnen 206
Erniedrigen, Priorität 105
Ersetzen, Systemkomponente 37
Erweitert, Dateiattribut 335
Erweiterung 35
Erweiterungsfeld 149
Erzwungen
-, Bearbeitung 268
-, Ereignis 310, 345
-, Hintergrundpriorität 101

Exec, Funktion 78
Exectype 79
Execution, Thread 355
Exit
-, Liste 248, 345
-, Routine 201
ExitList 248
-, Routine 201
Exklusiv
-, Benutzung 173, 197
-, Systemsemaphore 173, 201
Expertensystem 339
Explizit
-, Ein/Ausgabeoperation 233
-, Referenz 157
Extern 113, 148
-, Datenbedarf 122
-, Referenz 116
-, Unterroutine 115
Extra-RAM 181
Extrainstanz 207

FAIL, Anwort 304
Family API 31, 226, 319, 324
-, Anwendung 322
-, Lader 322
Far
-, Adresse 195
-, Aufruf 121, 137, 158, 221
-, return 179, 218
-, Unterroutine 217
Farbgebung 12
Farbkarte 306
FAT 26, 149, 155, 161, 256, 278, 331
Fehleingabe 47
Fehler 18, 235, 283
-, Ausgabeeinheit 68
-, Code 28, 78, 81, 178, 205, 220
-, Deklaration 311
-, Einheitentreiber 285
-, Fehlerbehandlung 316
-, GP 137, 199, 311, 347
-, INT 24 316
-, Klassifizierung 136
-, logisch 39
-, Mechanismus, GP 137
-, Meldung 68, 178, 243, 323
-, Programm 200
-, Segmentauslagerung 189
-, Segmentverschiebung 189
-, Suche 34, 45
-, Timeout 279
-, ungenügende Speicherkapazität 27
Fenster 238
-, Nummer 215
-, Technik 111, 223
Festlegen, Umfang 36
Festplatte 7, 187
FIFO 65, 211, 289
File allocation table 149
File Handle 59
Filtermodus 45, 345, 350
FIND 318
First in first out 211, 289
Flagge 57, 130, 193, 287, 296, 309
-, Argument 36
-, Freigabe 309
-, Ja/Nein 218
-, Register 217
-, Wort 206, 219
Flat Model 336
Flexibel, graphisch,
Benutzerschnittstelle 223
Flexibilität 34, 63, 137
-, Betriebssystem 37
Fließkommaregister 90
Fließkommazahl 158
Flüchtig, Speicher 161
Format 149, 152
Formatieren, MS-DOS-Diskette 259
Formatierfunktion 8
fprintf 60, 68
Fragmentierung 255, 330
Frame-Stack 355
Frei, Speicher 27, 182
Freigabe
-, Aufruf 172
-, Instanzdatensegment 132
-, interne Flagge 309
-, Prozeß 206
-, Ressource 275
-, Semaphore 132, 172, 194, 199, 274
-, Speicherbereich 18
Frontend, Treiber 139
fscanf 60, 68
Funktion
-, Bündelung 137
-, Exec 78
-, privilegiert 313
-, VIO, INT-10 ähnlich 227

GDT 123, 137, 157, 285, 345
-, Selektor 159
Gebunden, CPU 105
Gegenseitig, Blockieren 208
Gekennzeichnet, auslagerbar,
Systemsegment 177
Gelöscht, Pufferinhalt 73
Gemeinsam
-, benannter Speicher 193, 346
-, Hauptspeicher 133
-, Objekt 124

-, Pure-Codesegment 142
-, Ressource 129, 141
-, Speicher benennen 161
-, Speicherbereich 154
-, Speichersegment 161
-, Zugriff 142
Genauigkeit 279
General Protection 47
Geöffnet
-, Datei 70
-, Dateinummer 58
Gerät 8, 67, 264
-, Fehler 283
-, Geräteunabhängigkeit 11, 224
-, Überwachungsprozeß 265
-, Zugriff 305
Gesamtdatenbestand 220
Gesamtlösung, integriert 48
Gesamtprozeß, beenden, Form 98
Gesamtsystem 53
Geschlossen
-, Laufwerksklappe 299
-, Pipe 206
-, System 37, 139, 346
Geschützt
-, Betriebssystem 38
-, Ressource 200
-, Umgebung 12, 20
Geschwindigkeit 92, 119
-, CPU 279
-, Ein-/Ausgabe 257
-, Messung 194
-, Programmwechsel 54
Gesetzt, Semaphore 275
Gesperrt, Datei 303
GetDosVar 298
Gewährleisten, Datenaustausch 22
Gewichtig, Prioritätssteigerung 105
Giveaway shared memory 161, 193
Gleichartig, Softwareumgebung 20
Gleichzeitig
-, Aktualisierung 94
-, Ausführung 94
-, Programm 55
-, Projekt bearbeiten 16
-, Zugriff 28
Gleichzeitigkeit 56
Global
-, Daten 127
-, Datensegment 127, 131, 140, 346
-, Deskriptortabelle 123, 346
-, Infosegment 298
-, Initialisierung 131
-, Untersysteminitialisierung 346
GP, Fehler 47, 137, 160, 173, 199, 311, 347
Graphik 12, 226
-, VIO 228
Graphikanwendung 225
-, hochauflösend 241
-, Umschaltung 225
Graphikkarte 306
Graphikmodus 226
Graphikorientiert, Betriebssystem 109
Graphikprogramm 48
Graphiktreiber, geräteunabhängig 11
Graphisch
-, Anwendung, Unterstützung 12
-, Benutzerschnittstelle 12, 330
-, flexible Benutzerschnittstelle 223
-, Schnittstelle 23, 318
-, Standardoberfläche 223
Grenze 94
Groß
-, Datenmenge 15
-, Laufwerk 254
-, Pufferbereich 78
-, Speichersegment 164
Großbuchstabe 149
Großrechnerumgebung 183
Großschreibung 153
Großvaterprozeß 57, 76, 347
Gruppenname 334
GUI 109
Gültigkeitsbereich 97

Halbduplex 204
Handle flushing 276
Hardware
-, EGA-kompatibel 231
-, Fehler 91, 230, 283, 299, 303, 347
-, -, Daemon 233, 257, 301, 316, 347
-, Interrupt 17, 293, 306
-, Schutzleistungsmerkmal 160
-, Vorrichtung 157
Hauptspeicher 89, 133, 175
-, Kapazität 175
-, Virtualisierung 17
Header 321
Heap
-, Algorithmus 169
-, Header 171, 174
-, Objekt 169
-, Struktur 174
Hilfsmittel, Ausführung, Anwendungsprogramm 25
Hilfsthread 268
Hintereinanderschaltung, Zugriff 269
Hintergrund 16, 91, 101, 318
-, Aktualisierung 235
-, Ausgabe 228
-, Bildschirmgruppe 229, 318

-, Eingabe 228
-, Kategorie 347
-, Modus 229
-, Priorität 282
-, -, erzwungen 101
-, Thread 105
History, Eintrag 76
Hochauflösend
-, Bildschirmdarstellung 15
-, Graphikanwendung 241
Hochentwickelt, Sprache 221
Höchstleistungsprozessor 336
Huge
-, memory 164
-, Segment 347
-, Speicherobjekt 165

IBM XT/AT 6, 8
ID, Thread 356
Identifikation, Benutzer 335
IFS 150, 347
Ignorieren 274, 299
Ikone 111, 223, 228
Implementation 64
Implementationsdetail 36
Implementierung, Detail 135
Implizit
-, Referenz 157
-, Speicheranforderung 178
INCLUDE-Datei 191
Indentifikationscode 215
Independent Software Vendor 252
Indirekt, Aufruf 119
Industriell, Revolution 3
Information 184
Informationssegment 184, 282
Informationsstruktur 36
Infoseg 184, 282, 298
Infosegment, global 298
INIT-Zeichenkette 191
Initialisierung 74, 205
-, global 131
-, Information 191
-, Modul 133
-, Programm 190
-, Prozeß 130, 162
-, Routine 275
Initiierung
-, Deamon 134
-, Kindprozeß 66
-, Operation 295
-, Thread 99
-, verborgenes Programm 81
Installable file system 150
Installierbar, Daeiverwaltungssystem 150, 336, 347
Instanz 278, 288, 296
-, Daten 126
-, Datensegment 126, 131, 140, 348
-, -, Freigabe 132
-, Initialisierung 131
-, mehrere 207
-, Parameter 206
-, Pipe 210
-, Untersysteminitialisierung 348
-, Zählargument 207
INT 10/16-Schnittstelle 225
INT 21 300, 313
INT-10-ähnlich, VIO-Funktion 227
Interaktiv 101, 226
-, Anwendung 242
-, Benutzer 20, 81, 91
-, Eingabe, Ausgabe 241
-, Kategorie 348
-, Operation 61
-, Programm 237
-, Prozeß 104
-, Task 104
-, Vordergrund-Thread 106
-, Vordergrundanwendung 107
Intern
-, Betriebssystemstruktur 35
-, Fragmentierung 255
-, Softwareschnittstelle 29
-, Speicherbereich 99
Interprocess communication 193
Interrupt 10 225
-, ausgelagerter Code 177
-, -, Routine 263, 294
-, Einheit, gesteuert 139, 287
-, Pseudo 219
-, Rate, hohe 337
-, Service, Thread 296
-, Thread 291
-, Vektor 160
-, -, Peripherieeinheit 26
-, -, verbiegen 263, 315, 325
-, -, Tabelle 160
-, Zeitpunkt 298
IOCTL
-, Aufruf 290
-, Schnittstelle 297
IPC 80, 93, 134, 162, 193, 317, 348
-, Verknüpfung 216
-, Flaggensignal 221
IRET 217
ISAM, Verzeichnis 134
Isolierung 20, 348
ISV 252

Ja-Flagge 218
Journaldatei 62, 68
-, Aktualisierung 132

Kalenderprogramm 53
Kalkulationstabelle, Datenformat 135
Kapazität 178
Kapazitätsverlust 255
Kategorie, interaktiv 348
KBD 110, 241, 306, 349
Kennummer 75, 97
Kernel 14, 36, 69, 174, 194, 219, 349
-, API 37
-, Architektur 309
-, blockieren 301
-, Code 96, 137, 177
-, Modus 194, 286, 300, 309, 311
-, Schnittstelle 136
Kill 173
-, Signal 275
Kind
-, Befehlsunterbaum 248
-, Prozeß 43, 57, 189, 349
-, -, Beendigung 244
-, -, direkt 76, 85
-, -, initiieren 66
-, -, Priorität 108
-, -, Speicherkapazität 70
-, -, Thread 310
-, Task 87
Klassifizierung, Fehler 136
Klein, Speicherbereich 178
Kleinbuchstabe 149, 153
Klientenprozeß, Selektor 298
Klientenselektor 163
Koexistenz, Threads 89
Kommunikation 198, 285, 297
-, direkt 67
-, Programm 93
-, Prozeß 21, 193, 204, 349
-, Schnittstelle 237
-, Thread 198
Kompatibel, Betriebssystem 329
Kompatibilität 152
-, Box 313
-, Ebene 26
-, Modus 25, 319
-, Presentation Manager 112
-, Real-Modus 314
-, VIO 112
Komponentensegment 164
Konfiguration 66
Konfigurationsdatei 120
Konfigurationsinformation 191
Konsole, Verwendung 94
Kontextumschaltung 121, 270, 310, 349
Kopie
-, byteorientiert 260
-, Umgebungsstring 82
Korrektur, Eingabe 47
Kreativität 3
Kritisch, Bereich 97, 174, 269, 317, 349
-, Schutz 196
-, Semaphore 198
-, Unterbelegungspaket 171
-, wiederherstellen 275
-, Zugriff 269

Lade
-, Anweisung 176
-, benötigter Speicherbereich 18
-, Bereich 176
-, Bildschirmgruppe 302
-, dynamisch, Verknüpfung 115, 131
-, Prozeß 180
-, Segmentregister, Anweisung 178
-, Sektor 123
-, Speicherbereich 233
Lader 322
Ladevorgang 119, 178
LAN 16, 152
Lang, Antwortzeit 180
Längenangabe 159
Längenargument 78
Large-Segment 336
Laserdrucker 4
Laufbalken 223
Laufwerk 187
-, CD-ROM 336
-, Controller 293
-, Größe 254
-, Klappe 258, 299
-, logisch 255
-, orientiert, Operation 260
-, WORM 150, 336
Laufzeit
-, Bibliothek 170
-, dynamisch, Verknüpfung 119, 131, 148
LDT 123, 137, 157, 285, 350
-, Bereich 125
-, -, reservieren 164, 353
-, Eintrag 123
-, Selektor 124, 159
-, Tabelle 160
Leerlaufprozeß 143
Leistungsdurchsatz 209

Lesen
-, Nummer 65, 204
-, Thread 268
-, vergleichen 185
-, Vorgang 209, 212
-, Zeichen 88
LIB-String 191
Linken 38, 113
Liste, Arbeitsverzeichnis 43
Load on demand 145
Local area network 152
Local Descriptor Table 123
Lock 231
Logisch
-, Einheit 188, 350
-, Fehler 39
-, Laufwerk 255
-, Verzeichnis 188, 350
Lohnabrechnung 3
Lokal
-, Deskriptortabelle 350
-, Stapelvariable 95
-, Variable 94
-, Verwaltung, Segment 171
Lokalisierung, Fehlersuche 45
Lokalität, Referenz 182
Löschen
-, Bildschirmdarstellung 230
-, datenträgerorientiert 276
-, nummernorientiert 276
Löschoperation 276
LRU-Prinzip 176
LSL 138

Mail-Anwendung 55
Mainframe-Computer 55
Makroassembler 242
Manipulieren, Interruptvektor 26
Markiert, Beendigung, Prozeß 88
Markt, potentiell 252
Marktdurchdringung 30, 329, 339
MASM 242, 245
Massensoftware 251
-, Entwicklung 251
Massenspeicher 330
-, Zugriff 102
Maus 110
Mauseingabe 104
Mausschnittstelle 223
Maximal
-, Pufferbereich 78
-, Speicherbereich 27
Mechanisieren, Arbeit 3
Mechanismus
-, byte-orientiert 203
-, datagram-ähnlich 208
-, IPC, stromorientiert 202
Medium, austauschbar 257
Mehr-CPU-System 95
Mehrbenutzerumgebung 182, 331
Mehrfach
-, Nennung 188
-, Thread 317
Meldung 211
-, Anzeige 303
-, Bereich 213
-, höchste, Priorität 212
-, Inhalt 213
-, Modus 203
-, Orientierung 215
-, privat 29
-, Strom 210
-, Warteschlange 107, 213
Memory
-, Manager 157
-, Mapped 306
-, overcommit 177
Menü, Drop-Down 223
Messen, Zeitintervall, synchron 281
Mikroprozessor 5, 54, 314
Minimieren, Speicheraufsplittung 314
Minutenangabe 185
Modul 114
-, Initialisierung 133
-, Nebeneffekt 146
Modular, Systemaufbau 49, 55
Modulname 148
Modulnamenzeichenkette 116
Modus
-, Kernel 286
-, privilegiert 352
-, Real-Bildschirmgruppe 110
-, Umschaltung 337
Monitor, Thread 107
Monitordatenbearbeitung 107
Monitoring 284
Monitorkette 268
Monitorprozeß, Reihenfolge 266
MOU 110, 350
MS-DOS 5, 8, 303
-, Diskette 259
-, Header 323
-, Umleitung 264
-, Version 252
MS-Net-Server 332
Multiprocessing-Umgebung 215
Multiprozessorsystem 93
Multitasking 8, 11, 16, 53
-, Betriebssystem 350
-, parallel 16, 54, 351
-, quasi-parallel 17

-, seriell 16, 54, 353
-, Untersystem 109
Multithread, Prozeß 142

Nachfolgesystem 339
Nachladen, Segment 178
Name
-, Bereich 154
-, Dynlink 148
-, Einsprungspunkt 148
-, Feld 149
-, Gebung 188
-, Generierung 152
-, Kollision 155
-, Zeichenkette, umfangreich 163
Nebeneffekt 26, 42, 81, 100
-, Dynlink-Modul 146
-, Unterstützung 46
Nein-Flagge 218
Netzwerk 4, 16, 29, 203
-, Anwendung 106
-, Architektur 209
-, Funktion 29
-, Installation 331
-, Meldung 55
-, Protokoll 93, 335
-, Server 150, 276, 303
-, Umgebung 208
-, Übertragung 208
-, Zugriff 152
Nichtdifferenziert, Strom, Byte 203
Nichtfiltermodus 45, 350
Nichtgeöffnet, benannt, Pipe 208
Nichtunterbrechbar, Systemaufruf 221
Nichtverfügbarkeit, Software 30
Niedrig
-, Dichte 259
-, Priorität 107, 268, 349
Null
-, Einheit 240
-, String 36
NUMADD 59
NUMARITH 60
Nummer 58, 257, 351
-, Mechanismus 58
-, Operation 260
-, Orientiert, Löschen 276
-, Schnittstelle, standardisiert 68
-, Tabelle 72
-, Zugriff 244
Nur-Lese 335
-, Speicher 47
-, Zugriff 184
Nur-Schreib-Register 231
Nutzung, gemeinsam, Pure-Codesegment 142
Nutzungsgrad, System 279

Oberfläche, Presentation Manager 224
Oberflächendarstellungsform 109
Objekt
-, Datei 115
-, Dateinamensbereich 154
-, Eigentümer 333
-, gemeinsam genutzt 124
-, Name 155
-, Namenargument 78
-, Namenspuffer 351
Off-line-Betrieb 43
Offen, System 37, 139, 351
Öffnen
-, Datei 137
-, Laufwerksklappe 258
-, Pipe 206
Offset 158
-, Adresse 124
-, Segment 159, 299
-, Selektor 299
-, Wert, skaliert 159
Operation
-, Ausgabe 257
-, Beendigung 292
-, blockorientiert 230, 261
-, dateiorientiert 260
-, Eingabe 257
-, Initiierung 295
-, interaktiv 61
-, laufwerksorientiert 260
-, Modi 24
-, Vervollständigung 292
-, zeichenorientiert 289
Optimal
-, Antwortzeit 106
-, Datenspeicherung 261
-, Fehlersuche 34
-, Paging 174
-, Parallelverarbeitung 292
-, Strategie 33
Optisch, Zeichenerkennung 4, 23
Originalbildschirmgruppe 302
Originalsegment 127
OS/2, Arbeitsweise 62
Overcommit, memory 177

Paging 174
Paket, integriert 48
Parallel, Multitasking 16, 54, 351
Parallelverarbeitung 292
Partition 254
Passwort, persönlich 334
PATH 190
PC, vernetzt 331

PC-Net-Server 332
Peek 212
Performance 12, 15
Periodisch, Zeitgebreinterrupt 279
Peripherieeinheit 150
-, direkter Zugriff 11
-, Interruptvektor 26
Personal Computer 5, 20
Pfadname 151
Pfadnamenschnittstelle 151
Physikalisch
-, Einheit 188
-, Speicher 351
-, Verzeichnis 188
PID 74, 76, 84, 162, 211, 351
Pipe 162, 244, 351
-, anonym 64, 83, 202, 341
-, API, benennen 204
-, benannt 64, 154, 209, 211, 342
-, benennen 194, 202
-, erneute Öffnung 206
-, geschlossen 206
-, Instanz 210
-, nicht geöffnet 208
-, Nummer 65
-, testen 210
-, Wiederverwendbarkeit 204
Pixel 224
Plattenoperation, Ausführung, Diskettenoperation 103
Plotter 55
-, Dynlink-Treiber 120
PM 225
Polling 281
-, Schleife 274
-, Wartevorgang 216
Port
-, asynchron, seriell 93
-, seriell 198
-, elektronisch 16
Preemptive 100
Preload 145
Presentation Manager 82, 104, 109, 168, 214, 223, 351
-, Kompatibilität 112
-, Applikation 227
-, Oberfläche 224
Priorität 44, 56, 100, 352
-, allgemein 101
-, Angabe 100
-, angepaßt 103
-, Änderung, Thread 108
-, Ebene 108
-, Gesteuert, Vorrang-Scheduler 56
-, Hintergrund 282
-, hohe 106
-, Kategorie, allgemein 101, 341
-, -, niedrig 268
-, niedrig 107, 349
-, Scheduler 89
-, Steigerung 105
-, Thread 100, 280
-, Vaterprozeß, Kindprozeß 108
-, Überprüfung 100
-, verändern 87, 105
-, zeitkritisch 101, 106, 358
Privat, Meldung 29
Privilegänderung 217
Privilege-Modus 13, 160
Privilegiert
-, Funktion 313
-, Modus 352
Problem
-, diagnostizieren 19
-, Simulation 19
Process identification 74, 351
Produktlinie 252
Program application interface 161
Programm
-, Abarbeitung 182
-, ablauffähig 168, 271
-, Aufruf 49
-, Ausführung 244
-, -, asynchron 79
-, -, Beginn 90
-, -, Prinzip 64
-, ausgelagert, Aktivierung 175
-, Beendigung 19, 217, 274
-, Code 217
-, Datei 190
-, Entwurfsphilosophie 33
-, Fehler 38, 200
-, gemeinsame Benutzung 118
-, gleichzeitig 28, 55
-, interaktiv 237
-, sendend 107
-, speicherresident 263
-, Spooler 106
-, unabhängig 48
-, verborgen, Initiierung 81
Programmiersprache, höher 114, 138
Programmierung, Einheitentreiber 287
Programmlader 26
Programmlogik, inkorrekt 47
Programmstart 80
Programmstruktur 33, 93
Programmteil
-, unzusammenhängend 184
-, versionsspezifisch 325
Programmwechsel, Geschwindigkeit 54
Projekt, gleichzeitig bearbeiten 16
PROMPT 191

Proportionalschrift 13
Protect-Modus 8, 24, 157, 189, 314, 352
Protokoll 193, 214
Prototypversion 78
Prozedur 113
-, Aufruf 209
Prozeß 57, 352
-, abgehängt 81
-, aktiv 133
-, Anfrage 119
-, Baum 240
-, Bearbeitung 265
-, Beendigung 76, 98, 130, 245
-, Bildschirmgruppe 237
-, dediziert 141
-, durchschreiben 277
-, dynamische Verknüpfung 121
-, Eingriffsmöglichkeit 81
-, Element 89
-, Initialisierung 130
-, interaktiv 104
-, Kommunikation 21, 193, 204
-, laden 180
-, markiert 88
-, Multithread 142
-, PID 76
-, Priorität 44, 108
-, Rückschaltung, Vordergrund 234
-, Schnittstelle 133
-, senden 211
-, Tastatur, verwenden 241
-, Tastenanschlag, übergeben 238
-, umschalten 16
-, Unterbaum 246
-, unterbrechen 97
-, vollständig, auslagern 175
-, vorzeitig beendet 215
Prozessor
-, Emulation: 8086/8088 24
-, Schnittstelle 134
Prüfen, Datenträger 276
Prüfsumme 141, 259
Pseudo-Interrupt 219, 295
Pseudoverzeichnis 155
Ptrace 305, 308, 312
Puffer 92, 130, 256
-, Bereich, maximale Größe 78
-, Inhalt, gelöscht 73
-, Länge 36
-, Löschen 278
-, STDOUT 73
-, Wiederverwendbarkeit 213
Pure, Segment 141

Quasi-parallel, Multitasking 17
Querverbindung, Anwendung 82
Queue-API 211

Rahmenbedingung 58
RAM 89, 194, 233, 338
-, Disk 187
-, Kapazität 178
-, Puffer 65, 92, 256
-, Semaphore 92, 96, 171, 194, 221, 272, 290, 352
-, -, Operation 195
RAS 185
READ 148
Real-Modus
Real-Modus 8, 24, 159, 177, 181, 226, 303, 353
-, Bildschirmgruppe 110, 313
-, Box 337
-, Kompatibilität 314
-, virtuell 308, 337
Rechenkapazität 56
Rechenleistung 330
Rechenzeit 12, 79, 94, 100, 108
Rechtschreibprüfungsprogramm 187
Redirector 29
Referenz
-, explizit 157
-, extern 116
-, implizit 157
-, Lokalität 182
Register 217
-, CS 158
-, Nur-Schreib 231
-, Inhalt 90
Registersatz 219
Reihenfolge, Monitorprozeß 266
Reliability 185
Relokationsberechnung 323
Remote
-, Operation 303
-, Prozeduraufruf 209
-, System 154
-, Zugriff 203, 330
Request-Paket 14
Reservieren, LDT-Bereich 123, 125, 164, 353
RESET 148
Resident, Speicher 30
Ressource 89, 101, 133, 269, 288
-, freigeben 275
-, gemeinsam 129, 141
-, geschützt 200
-, Manipulation 273
Return 221
-, far 179
Revolution, industriell 3

Ring 14, 24, 96, 194, 285, 300, 307, 353
-, 0-Code 138, 160, 220
-, 3-Code 220
-, Befehl 264
-, Privileg 307
-, Segment 179
-, Umschaltung 15, 121
RMSIZE-Grenze 181
Rollbalken 110
Rollen, Bildschirmdarstellung 230
ROM-BIOS
-, Bildschirmschnittstelle 224
-, Selektor 160
Routine
-, Adresse 119
-, bearbeiten, DosExitList 173
-, Dynlink 122, 344
-, -, ladedynamisch 131
-, -, laufzeitdynamisch 131
-, Initialisierung 275
-, interruptbearbeitend 263, 294
-, signalbearbeitend 172, 216, 248
Rückgabe 231
Rückgabecode 28
-, direkt, Kindprozeß 83
Rückgabeparameter 170
Rückkehrsequenz 265
Rückrufereignis 265
Rückschaltung, Prozeß 234
Rücksprung 199
Rundspruchverfahren 214

Sammelpuffer 277
Scanner 4
Schaubild 48
Scheduler 11, 44, 94, 100, 353
-, Priorität 89
-, Verteilungsalgorithmus 100
Schlange 294
Schleife, Polling 274
Schließen
-, Dateinummer 71
-, Einheit 70
Schlüssel, Verteilung 333
Schnell
-, Speichereinheit 191
-, umschalten, Anwendungen 16
Schnellstmöglich, Antwort 103
Schnittstelle 13, 58, 209, 223
-, bildschirmorientiert 318
-, Dateiverwaltungssystem 36
-, DevHlp 297
-, einheitlich 31
-, graphisch 23, 318
-, IOCTL 297
-, Kernel 136
-, Paket 71
-, Protokoll 23
-, Prozessor 134
-, Spezifikation 23
-, TTY 109
-, VIO 112
Schreibgeschützt 184
Schreibnummer 65, 204
Schreibvorgang 211, 259
Schriftart 13
Schutz 19
-, Codesequenz 230
-, Datei 331
-, Fehler, allgemein (GP) 341
-, Hardware 160
-, kritisch, Bereich 196
-, Modell 157
-, vollständig 337
-, Vorkehrung 288
Schwer, GP, Fehler 199
SCP-DOS 6
Segment 175
-, Arithmetik 158, 166
-, Auslagerung 95, 149, 189, 314
-, Basis 169, 175
-, fault trap 175
-, Huge 347
-, Large 336
-, lokal 171
-, nachladen 178
-, Offset 159, 299
-, Pure 141
-, Register 123, 158
-, Selektor 123, 159, 164
-, Selektorwert 298
-, Verschiebung 189
-, Zeiger 170
Sektor 254
-, laden 123
-, Beschränkung 254
Selektor
-, aufeinanderfolgend 261
-, Ausrichtung 262
-, Empfänger 213
-, gleichweit auseinanderliegend 167
-, Grenze 261
-, Klientenprozeß 298
-, LDT 124
-, Nummer, Warteschlange 291
-, Offset 299
-, ungültig 125, 141
-, Wert 141
Self Sizing 36

Semaphore 130, 193, 253, 272, 296, 317
-, Anforderung 194
-, Ereignis, mehrere 254
-, Freigabe 172, 194, 199, 274
-, kritisch, Bereich 198
-, Nummer 59, 195
-, RAM 92, 96, 290
-, setzen 275
-, verwaisen 199
-, warten 288
-, Wiedergewinnung 199
-, Zähler 198
-, Zuordnung 202
Senden 107, 211
Sensitiv, Daten 20
Separat, Liste, Arbeitsverzeichnis 43
Seriell
-, asynchron, Port 93
-, Multitasking 16, 54, 353
-, Port 198
Server 150
-, Benutzer 31
-, Datenbank 162
-, Ende 207
-, Gerät 276
-, Instanz 207
-, Prozeß 133
-, System 332
-, Thread 208
Serviceability 185
Serviceanforderung, benennen 162
Serviceroutine 237
Session Manager 244, 354
SetSignalHandler 88
Setzen, Prozeßpriorität 108
SFT 72, 355
Shared memory 161, 193
Shell 238
Sicherheitsabfrage 27
Sicherheitssystem 93
Sichern, Bildschirminhalt 229
SideKick 263
SIGCTLC 244
SIGINT 171
SIGKILL 88, 310
Signal 97, 132, 171, 194, 215, 245, 274, 308, 354
-, Adresse 221
-, Anforderung 221
-, ankommend 311
-, Art 132
-, Bearbeitend
-, entdecken 220
-, Ereignis 219
-, Gabe 354
-, Handhabung 147
-, KILL 275
-, Mechanismus 264
-, Routine 172, 216, 248, 275
-, Unterbrechung 218
-, Übermittlung 199
-, Verzögerung 172
Signalbehandlung, Thread 200
Signalisierung, Zustand 194
SIGTERM 248
Simulation 339
-, auftreten, Probleme 19
Simultan, Abarbeitung 91
Singletasking-System 20, 39, 182, 300, 354
Sinnbild 223
Situation, unlösbar 19, 172
Skaliert, Offsetwert 159
Software
Software
-, Ausbau 49
-, Design 33
-, Entwurf 33
-, geschützte Umgebung 20
-, Modularität 49
-, Nichtverfügbarkeit 30
-, Paket 85
-, Schnittstelle, intern 29
-, Schutz 20
-, Standard 319
-, Tools 34, 48
-, -, Approach 354
Sonderbildschirmgruppe 81
Sortierung 21, 55
Speicher
-, 1-Wait-State 99
-, Adressierung 40
-, Anforderung 18, 27
-, -, implizit 178
-, -, Umfang 170
-, Aufsplittung, minimieren 314
-, Aufteilung, System 315
-, Auslagerung 18, 44, 175, 233, 354
-, Ausnutzung 27
-, Baustein 183
-, Belegung 24, 99, 137, 189, 313
-, Benennen, gemeinsam 193
-, Bereich 78, 216
-, -, adressierbar 34
-, -, Auslagerung 17, 175
-, -, belegt 18
-, -, benötigt 18
-, -, frei 182
-, -, freigeben 18
-, -, gemeinsam 154
-, -, intern 99

-, -, laden 233
-, -, maximal 27
-, -, statisch 95
-, -, unbenutzt 40
-, -, unterer 182, 314
-, -, zugreifen 17
-, Dateiname 152
-, Daten 261
-, Datenstruktur 269
-, Deskriptortabelle 285
-, Einheit 38, 191
-, flüchtig 161
-, gemeinsam, benennen 161
-, -, benutzt 193
-, intensiv 184
-, Kapazität 12, 27, 70, 99, 175, 181
-, Name 163
-, Objekt 169, 213
-, -, huge 165
-, physikalisch 351
-, Platz, Thread 99
-, resident 30, 263
-, Schutz 38, 169
-, Segment 90, 161
-, Seitenmanagement 174
-, Stelle 47, 159, 182
-, Unterbelegung 169, 174, 213, 355
-, Verdichtung 99
-, Verwaltung 11, 17, 44, 109, 157, 161, 183, 338, 355
-, Verwendung 44, 177
-, virtuell 18, 357
-, Zuweisung 178
Sperraufruf 172
Sperren
-, Bildschirm 230
-, Datensatz 44
Spooler 55, 106, 200, 209
Sprache, hochentwickelt 221
Stabil, Umgebung 34, 38, 42, 264
Stammverzeichnis 7, 256
Standard 251
-, Akzeptanz 251
-, Ausgabe 60, 73
-, Betriebssystemschnittstelle 26
-, Bibliotheksmodul 191
-, Computerumgebung 339
-, Dateinummer 58, 64
-, Fehlereinheit 68
-, Laufzeitbibliothek 114
-, Nummernschnittstelle 68
-, Routine 114, 317
-, Software 48
-, Umgebung 12, 31
Stapel 91, 294
-, Anforderung 99
-, Bereich 91
-, Segmentfehler 91
-, Thread 90
-, Umschaltung 217
-, Variablen, lokal 95
-, Verarbeitungsdatei 43, 64, 190, 238
Startoperation 295
Startsektor 259
-, Datenträgerkennummer 260
Statisch
-, Datenaktualisierung 95
-, Datenbereich 122
-, Speicherbereich 95
-, Verknüpfung 113, 355
Status 184, 294
Statusinformation 296
Statussystem 295
STD-Dateinummer 69
STDERR 59, 69
STDIN 59, 69, 112, 238
STDOUT 59, 69, 112, 238
-, Pufferung 73
Steigerung, Priorität 105
Steuertaste 45
STI 222, 297
String, beendet 36
Strom, Byte, nichtdifferenziert 203
Strommodus 205
Stromorientiert, IPC 202
Struktur
-, .EXE-Datei 321
-, Betriebssystem 35
-, Heap, Wiederherstellung 174
-, Prinzip 33
Subroutine 128
Subscreen group 185
Substitutionsträgerkennummer 259
Subsystem 128
Subtask 57
Suche, Fehler 45
Suchvorgang 294
Suchzeiger 72
Supercomputer 339
SWAPPER.DAT 180
Swapping 44, 175
Symbol
-, definiert 113
-, Dynlink 128
Synchron
-, Ausführung 79
-, Zeitintervallmessung 281
Synergieeffekt 252
System 335
-, abschalten 187
-, Absturz 40, 46, 276
-, API 194

-, architektur 64, 139
-, Aufruf 96, 138, 219
-, -, blockierend 103
-, -, nichtunterbrechbar 221
-, -, Unterstützung 307
-, -, zurückkehren 88
-, Dateitabelle 72, 355
-, Debugger 81
-, Durchsatz 104, 176, 292
-, file table 72
-, Flexibilität 49
-, geschlossen 37, 139, 346
-, Hardware 26
-, Kernel 138
-, Komponente, ersetzen 37
-, Nutzungsgrad 279
-, offen 37, 351
-, Overhead 178
-, Programmierung 34
-, Prozeß-Aufruf 232
-, Remote 154
-, Ressource 157, 194
-, Schnittstelle 35, 138, 320
-, Schutz 19
-, Segment, auslagerbar, gekennzeichnet 177
-, Semaphore 59, 154, 173, 194, 197, 272, 275, 355
-, -, Freigabe 132
-, -, Operation 195
-, Sicherheit 20
-, Softwareumgebung 20
-, Speicheraufteilung 315
-, Sperre 18
-, Spooler 209
-, Stabilität 12, 307
-, Start 110, 284
-, Takt 17, 280
-, Uhr 160
-, Utility 79
-, Zeitgeberroutine 199
-, Zustand 19

Tabelle 12
Tabellenkalkulation 48
Tageszeit 318
Taktfrequenz 194
Task 57, 355
-, interaktiv 104
-, Kind 87
-, Komponente 296
-, Thread 294
-, Zeitpunkt 284, 289
Tastatur 160
-, Ausgabe 248
-, Benutzung 301
-, Bildschirm, Zugriff erben 243
-, Eingabe 103, 221, 237, 248
-, -, abfangen 42
-, Erweiterungsprogramm 46
-, Interruptvektor 263
-, Kontrolle 233
-, Makro 46
-, Modus 45
-, Monitor 47, 268
-, Prozeß, verwenden 241
-, Puffer 248, 313
-, Schnittstelle 223
-, Vektor, Verbiegen 41
-, Zugriff 239
Tastenanschlag, Prozeß 238
Tastendruck 42
Tastenkombination 245
Teillösung 55
Telekommunikationsanwendung 106
TEMP 191
Temporär, Umbenennung 67
Temporärdatei 191
Temporärspeicher 83
Testen, Pipe 210
Textverarbeitungssystem 235
Thread 56, 89, 97, 157, 356
-, abgebrochen 96
-, Ausführung 219
-, beenden 98
-, dynamische Verknüpfung 121
-, Execution 355
-, gleichzeitige Ausführung 94
-, Hintergrund 105
-, höhere Priorität 100
-, ID 356
-, Initiierung 99
-, Interruptservice 296
-, Kennummer 97
-, Kindprozeß 310
-, Koexistenz 89
-, Kommunikation 198
-, kritisch 272
-, lesen 268
-, mehrfach 317
-, Monitor 107
-, of Execution 89
-, Priorität 280
-, -, ändern 108
-, -, setzen 108
-, Signalbehandlung 200
-, Speicherplatz 99
-, Stapel 90
-, Task 294
-, Umschaltung 100
-, Unterbrechung 96

-, verborgen 98, 357
-, Vordergrund 106, 282
Time Slice 56
Timeout 93
-, Fehler 279
Token 333
Tool 48, 59, 64, 69
-, basiert, Architektur 193
Trace-Flagge 311
TREE 64
Treiber
-, Dynlink-Paket 120
-, Frontend 139
-, Datenträgerzugriff 177
Treiberbaustein 49
TTY, Schnittstelle 109

Übergabespeicher 161, 193, 356
-, Segment 162
Übergeben, Tastenanschlag 238
Überprüfung, Priorität 100
Übertragen 104, 163, 261
Übertragungsmechanismus 211
Überwachung 107, 263
Überwachungskomponente 268
Überwachungsprozeß 265
Umbenennung
-, Diskette 259
-, temporär 67
Umfang
-, festlegen 36
-, Speicheranforderung 170
Umfangreich, Namenszeichenkette 163
Umgebung 57, 324
-, 80286 174
-, angepaßt 12
-, Block 82
-, Änderung 43
-, geschützt 12, 20
-, Konfigurierung 82
-, Kontrolle 313
-, Liste 189
-, Multiprocessing 215
-, Segment 189
-, sicher 331
-, stabil 34, 38, 42, 264
-, String 44, 82, 187, 189, 356
Umleitung 264
Umschalten 271
-, Anwendungsprogramm 53
-, Bildschirmgruppe 110
-, CPU 317
-, Graphikanwendung 225
-, Prozeß 16
-, Ring 15
-, Thread 100
Umschalttaste, Betätigung 42
Umwandlungsprozeß 37
Unabhängig
-, interaktiv, Task 104
-, Programm 48
Unautorisiert, Zugriff 154
Unbenutzt, Speicherbereich 40
Ungenügend, Speicherkapazität 27
Ungültig
-, Dateiname 47
-, Selektor 125, 141
-, Sonderbildschirmgruppe 81
-, Speicherstelle 47
UNIX 5, 312
Unlock, lock 231
Unlösbar, Situation 19, 172
Unterbaum 245
Unterbelegungsaufruf 174
Unterbelegungspaket 169, 173
-, kritischer Bereich 171
Unterbildschirmgruppenwert 185
Unterbrechung 97, 318
-, Adresse 217
-, Arbeitsablauf 199
-, Blockierung 310, 356
-, Punkt 308
-, Signal 218
-, Stelle 218, 222
Unterdrückung, Nebeneffekte 46
Unterprogramm 60
Unterroutine 94, 113, 128, 129
-, extern 115
Unterroutinenbibliothek 136, 287
Unterroutinenpaket 121
Untershell 238
Unterstützung
-, graphisch 12
-, Systemaufruf 307
-, Tastaturerweiterungsprogramm 46
Untersystem 71, 109, 128
-, Dynlink 344
-, Initialisierung, global 346
-, Multitasking 109
Unterverzeichnis 7, 189
Unzusammenhängend, Programm 184
Ursprungspriorität 147
Ursprungsprozeß 43
Utility 79, 318

Value, subscreen, group 185
Variable, lokal 94
Vater-Kind-Beziehung 57, 90
Vaterprozeß 57, 60, 189, 356
-, Priorität 108
-, Umgebungsstring 82
Vektor, Drucker 263

Verändern, CPU-Priorität 87
Verarbeitung, asynchron 92
-, Einheit 38
-, Geschwindigkeit 15, 330
-, Leistung 330
Verbergen
-, Detail 216
-, Interrupt 325
-, Interruptvektor 263, 315
-, Programminitiierung 81
-, Thread 98, 357
Verbiegen, Tastaturvektor 41
Vereinzelung 348
Vererbbar, nicht 69
Vererbungsarchitektur 63
Verfügbar, Speicher 27
Verfügbarkeit, API 329
Vergeben, Datenträgername 258
Vergleichen, lesen 185
Verkapselung 63
Verknüpfung
-, dynamisch 15, 113, 224, 323, 343
-, -, Laden 115
-, -, Laufzeit 119
-, IPC 216
-, Prozeß 113
-, Routine, dynamisch 91
-, statisch 113, 355
Vernetzt, PC 20, 331
Verringerung,
Ausführungsgeschwindigkeit 19
Version
-, Aktualisierung, Drittthersteller 37
-, MS-DOS 252
-, Spezifisch, Programmteil 325
Verteilung, Schlüssel 333
Verteilungsalgorithmus 100
Vervollständigung, Operation 292
Verwaisen, Semaphore 199
Verwaltung, lokales Segment 171
Verwenden
-, Datei 44
-, exklusiv 173
-, Konsole 94
-, Speicher 44
-, Tastatur 241
-, virtueller Speicher 177
Verzeichnis
-, Hierarchie 7
-, ISAM 134
-, logisch 188, 350
-, Name 189
-, physikalisch 188
VIO 71, 110, 129, 223, 267, 297, 306, 357
-, Aufruf 228
-, Graphik 228
-, Kompatibilität 112
-, Schnittstelle 112, 225
VioGetBuf 229
VioModeWait 234
VioSavRedrawWait 232
VioScrLock 229
Virtualisierung 13, 17, 43, 357
Virtuell
-, Anzeigeeinheit 13
-, Bildschirmgruppe 238
-, Real-Modus 308, 337
-, Speicher 18, 338, 357
-, Verwendung, Speicher 177
Vollduplex 204
Vollständig
-, Prozeß auslagern 175
-, Schutz 337
Volume flushing 276
Von-Neumann, Architektur 38
Vordergrund 91, 101
-, Anwendung, interaktiv 107
-, Bildschirmgruppe 301
-, Kategorie 357
-, Modus 228
-, Prozeß 233
-, Rückschaltung 234
-, Thread 282
Vorrang-Scheduler 56
Vorzeitig, beendet, Prozeß 215

Wahlfrei, Zugriff 205
WaitOption 84
Warten, Semaphore 288
Warteschlange 132, 211, 357
-, Abarbeitung 212
-, Meldung 107, 213
-, Selektornummer 291
Warteschleife 92, 287
Wartevorgang, Polling 216
Wartezeit 210
Wechsel, Datenträger 257
Werkzeug 59
Wert, Timeout 280
Wiedergewinnung, Semaphore 199
Wiederherstellung, Heap, Struktur 174
Wiederholen 299
Wiederverwendbarkeit
-, Pipe 204
-, Puffer 213
Working-Set 183
WORM-Laufwerk 150, 336
Worst-Case-Fall 179
Wörterbuchdatei 187
Write once read many 150

Writethrough 276
WYSIWYG 13, 243

XENIX 5

Zähler 197
Zählsemaphore 197
Zeichen
-, EOF 72
-, Erkennung, optisch 4, 23
-, lesen 88
-, Modus 226
-, orientiert 73, 112, 225, 289
Zeichnung 12
Zeigeinstrument 110
Zeilenorientiert, Schnittstelle 109
Zeiteinheit 56, 357
Zeitgeber, Auflösung 280
Zeitgeberinterrupt, periodisch 279
Zeitgeberroutine 263, 279
Zeitintervall 105, 279
-, synchron messen 281
Zeitkritisch
-, Anwendung 163, 282
-, Priorität 101, 106, 358
Zeitpunkt, Task 284, 289
Zeitscheibe 54
Zentraleinheit 17
Zerlegen, Dateiname 36
Ziel 4, 11, 53
Zielelement 216
Zufallsfaktor 94
Zufallszahl 258
Zufallszahlengenerator 258
Zugreifen, Speicherbereich 17
Zugriff 162, 196
-, Berechtigung 202, 243, 309
-, Bildschirm 239
-, Datei 28
-, Datenstrom, Einheit 264
-, direkt 11, 23, 242
-, erben 243
-, Erlaubnis, abtreten 239
-, gemeinsam 142
-, Geräteeinheit 305
-, gleichzeitig 28
-, Konflikt 229, 301
-, kritischer Bereich 269
-, Liste 332, 335
-, Massenspeichereinheit 102
-, Nummer, ausüben 244
-, Recht 334
-, Schutz 154
-, Tastatur 239
-, Treiber, Datenträger 177
-, unautorisiert 154
-, Vergabe 239
-, Versuch 28
-, wahlfrei 205
Zuordnung, Semaphore 202
Zurückkehren, Systemaufruf 88
Zusammengehörig, Speicherstelle 182
Zustand, Signalisierung 194
Zuteilung, Rechenzeit 94
Zuweisung, CPU-Zeit 44
Zwei, Speicheranforderung 182
Zwischenspeicherung, Daten 152

INFO-KARTE **Bitte informieren Sie mich über die neue Vieweg-Infoware**

Ich habe einen Mikrocomputer:

Typ: _____ Hersteller: _____

Ich benutze den Mikrocomputer hauptsächlich privat ☐
hauptsächlich beruflich ☐

Vorwiegend wende ich den Mikrocomputer in folgenden Bereichen an:

_____

Diese Karte entnahm ich dem Buch: **Letwin, Inside OS/2**

Meine Buchhandlung: _____

Gleichzeitig bestelle ich: ____ Ex. _____

____ Ex. _____

Anschrift:

_____
Name Vorname

_____
Beruf/Funktion Straße

_____
PLZ Ort

Gordon Letwin

INSIDE OS/2

Beschreibung, Einsatz, technische Details

Gordon Letwin ist „Chief Architect of Systems Software" bei Microsoft. Als Entwickler von OS/2 kennt Letwin daher das neue Betriebssystem mit all seinen Vorzügen und Raffinessen (leistungsfähige Graphik, echtes Multitasking etc.). Insbesondere kann der Autor detailliert Auskunft geben über die besonderen, z. B. prozessorbedingten Unterscheidungsmerkmale von OS/2 gegenüber dem „alten" Betriebssystem MS-DOS.

Das Buch beschreibt die wesentlichen Elemente von OS/2. Dabei versteht es der Autor, die größeren Zusamenhänge, die zu der Entwicklung einzelner Features von OS/2 geführt haben, verständlich darzustellen. Darüber hinaus werden die technischen Details, die für jeden ernsthaften Programmierer von Belang sind, präzise und eingängig erläutert. Beispielsweise geht es um:

- Multitaskingabläufe
- VIO Benutzerschnittstelle
- Dynamisches Linken
- Speicherverwaltung
- Presentation-Manager
- Einheitentreiber

Ein kompetentes Buch, von dem OS/2 Insider für künftige Insider geschrieben.

ISBN 3-528-04628-7

Microsoft PRESS

VIEWEG®